
Adam Zalepa

ASSEMBLER

Programming base

Poland 2018

ISBN 978-83-947369-9-6

Wydawca:

A2

All names and trademarks used in the book belong to their owners and have been used for informational purposes only. Reproduction in whole or in part without the Publisher's permission is prohibited. The author and the publisher are not responsible for the consequences resulting from the use of information and programming techniques in this book.

**(C) Copyright by
A2 & Adam Zalepa**

Łódź 2018

All rights reserved

Printed in Poland

*Computers do not invalidate
the multiplication table.*

- Aleksander Kumor

TABLE OF CONTENTS

Introduction	11
BASICS	13
Operational memory	15
RAM memory	16
ROM memory	17
Memory addresses	18
Size units	19
Number systems	22
Hexidecimal system	22
Binary system	24
Processor registers	27
Multitasking	29
EDITOR	31
Starting from the floppy disk	33
Installation on hard disk drive	43
First steps	55
Default parameters	59
Memory	59
Working directory	62
Typing and testing the program	65

TABLE OF CONTENTS

Removing the program	72
Disk operations	74
Work on multiple files	79
Block operations	82
Searching the program	85
Settings	88
Display mode	90
Selection windows	92
Settings related to Workbench	94
Assembler settings	96
 FIRST PROGRAM	 101
Numbers	103
Variables	104
Comments	105
Labels	107
Stack pointer	109
Instructions and mnemonics	110
Operators	111
Executing the program	112
 SIMPLE OPERATIONS	 113

TABLE OF CONTENTS

Adding values	115
Subtracting values	118
Values multiplication	120
Dividing values	122
Single bit operations	124
Storing address in the register	127
Logical functions	128
Copying data	136
Increment and decrement	138
Clearing the contents of registers	140
Loops and jumps	142
Conditional statments	146
CMP instruction	146
BEQ instruction	147
BNE instruction	148
TST instruction	149
BPL instruction	150
BMI instruction	151
Flags	152
Carry flag	153
Overflow flag	155
Zero flag	156

TABLE OF CONTENTS

Negative flag	157
Extended flag	158
ADVANCED OPERATIONS	159
Converting numbers into text strings	161
Converting text strings to numbers	167
Moving bits	170
Attaching source files	174
OPERATING SYSTEM	177
Calling libraries	182
Program initialization	186
Reserving the memory	187
Opening the window	190
Input/output operations	198
Keyboard support	199
Disk operations	207
Reading files	208
Saving files	210
Removing files	214
Renaming files	215
Reading the directory using AmigaDOS commands	216

TABLE OF CONTENTS

Reading the directory without using AmigaDOS	221
Selection windows	233
Event handling	238
HARDWARE REGISTERS	241
Mouse and joystick support	244
Ending	249
Appendix A: Library table	251
Appendix B: Registers list	277
Appendix C: Keyboard shortcuts	290
Appendix D: Command and functions	303

INTRODUCTION

Assembler is a low-level programming language. Thanks to it, we can influence the behavior of the computer to the greatest extent, because it is the language closest to the machine code, ie the sequence of instructions that the processor can execute. Other languages, such as Basic or C, must first be "translated" into machine code. So you can say that Assembler is the only language directly understandable to the processor.

The great advantage of a machine language is that programs can run much faster. With an interpreter like Basic, each line must be interpreted before it is executed, which requires a lot of time. The machine language has access to all the functions of the computer because it is a native language for the computer. Unfortunately, this way of writing programs requires much more time to spend than learning AMOS or Blitz Basic. It is also much more difficult due to the fact that we must imagine the way the processor and its memory cells work.

In the book, I present the basic instructions and symbols that may seem very complicated. However, they are essentially abbreviations of English words. In addition, they are easy to remember and do not require entering too many values, such as some commands in the AmigaDOS window.

The examples in part have been taken from working fragments of programs or documentation included with Assembler editors. They have been modified and provided with an additional comment so that the reader can easily get an idea of the operation of individual functions.

However, it should be taken into account that these are only outlines showing the process of creating a program. In most cases, they should not be entered into the editor, but on their basis, you should create their own algorithms.

The series "Programming base" discusses only a section of the steps to be learned in order to write a ready-made big program. However, I take the position that the principle should be a solid learning of the basics, and only then we should go to more advanced functions.

That is why a second series of books will also be created, which will speak about the same programming languages, but in a big picture. In the case of Assembler, I will not explain all the details of using a machine code, but I will present algorithms with the help of which we will write a program using Amiga chipset or more elements of the graphical interface. In addition, several longer programs will be discussed along with a detailed list of operations performed in the computer's memory - step by step.

- Łódź, January 2018 r.

BASICS

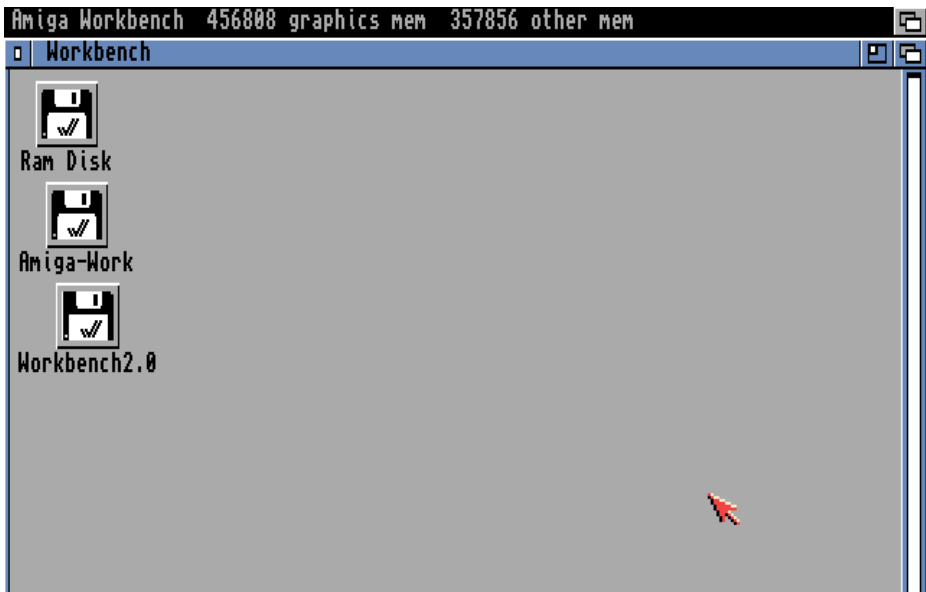
OPERATIONAL MEMORY

Before we start writing a program in machine language, we need to know the basic resources available in the computer. Memory is one of the most important.

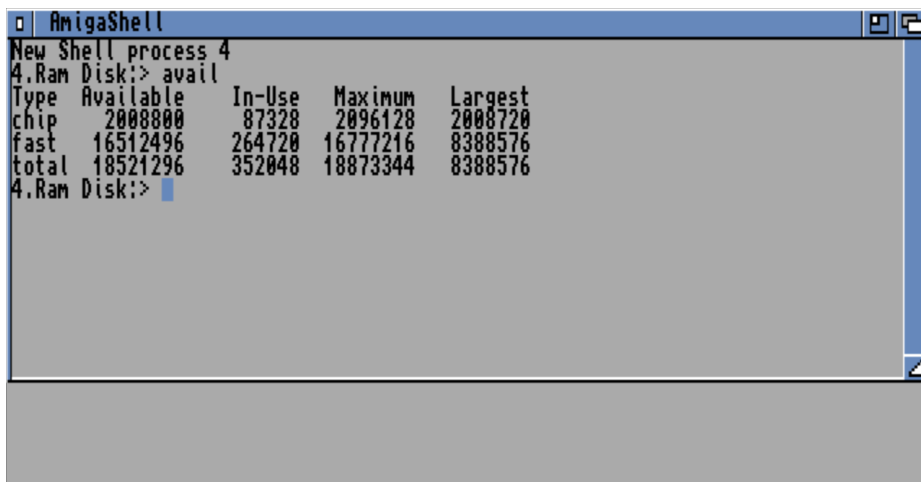
- RAM memory

RAM memory allows you to place information and then remove it at any time. It stores data only when the computer is turned on.

Amiga memory is divided into two parts called Chip or graphic memory and Fast, in the system defined as "other memory". You can check the available amount of memory by observing the Workbench title bar:



or using the AVAIL command in the AmigaDOS window:

A screenshot of the AmigaShell window. The title bar says "AmigaShell". The text inside the window shows a new shell process starting and the output of the "avail" command. The output is a table with four columns: Type, Available, In-Use, Maximum, and Largest. The rows are for chip, fast, and total memory.

```
New Shell process 4
4.Ram Disk:> avail
Type Available In-Use Maximum Largest
chip 2008800 87328 2096128 2008720
fast 16512496 264720 16777216 8388576
total 18521296 352048 18873344 8388576
4.Ram Disk:>
```

The computer motherboard has many chips that can run almost entirely independently, thus reducing the CPU load. Each of these chips also uses memory for action.

Graphical and audio data must be stored in the Chip memory. On the other hand, only the processor has access to the Fast memory, which increases the speed of operations. Amiga can work without the "other" memory, but when it is installed, accelerates computer about two times compared to a hardware with Chip memory only.

Remember that various Amiga models have limitations in the amount of graphics memory. We can have a maximum of 2 MB of Chip memory, but the standard Amiga 500 has only 0.5 MB, so only 512 kilobytes. However, Fast memory can be much more, from a few to even several hundred megabytes.

- ROM memory

The second type of memory - ROM - can store data even when the computer is turned off. However, the data in this case can only be read. In Amiga, the ROM memory is mounted on the motherboard and is called Kickstart.

After running, it performs a series of diagnostic and system tests, and then the basic components of the operating system are initialized. The connected boot devices are checked and the system boot is performed using the device with the highest priority - floppy disk drive or hard drive.

Kickstart contains basic components of the operating system such as libraries and component drivers on the computer motherboard. Versions for different Amiga models have characteristic parts that may not be present in other editions, even the same version of Kickstart. A good example is the "scsi.device" device that supports the built-in IDE controller. It has many versions differing in many features, and in some versions Kickstart does not exist at all.

It is worth noting that the first Amiga 1000 model required loading the Kickstart content using the included floppy disk. Up to version 2.0 it had a size of 256 kilobytes, and then it was expanded to 512 kilobytes.

MEMORY ADDRESSES

Different Amiga models can be equipped with different amount of memory. In order to get access to it, we need to save the value in a specific memory location. Therefore, we must use so-called addresses or otherwise - cells that determine the position in the computer's memory.

A memory address is, theoretically speaking, a unique memory cell identifier in which a certain amount of data can be stored for later use. A single address contains a byte, but in order to save more information, data can be saved in many addresses in succession or in a different order. The division of computer memory into addresses is necessary in order to find the information you need.

Different memory cells may have different functions, and the description of the entire available area is called the memory map. It is a complicated structure, which is why we will discuss it in parts, presenting specific applications. The map of Amiga's memory has been described in many books, but it will be useless for people who do not know how to use different memory cells.

That is why it can be a supplementary reading. However, the most important thing is to understand the mechanisms of direct impact on the behavior of the computer, which we achieve by saving and reading numbers that seemingly have no greater value.

Therefore, let's start with the basic information about numbers and other characters that can be found when entering and analyzing the machine code.

SIZE UNITS

The standard size in which memory is expressed is a kilobyte. One kilobyte consists of 1024 bytes, not 1000, as you might expect. This system results from the binary mode of computer operation, where the values are given as the power of the number 2. And yes, 1024 bytes are the result of the calculation:

2¹⁰

The same rule applies to larger units such as megabyte and gigabyte, where the first one means 1024 kilobytes, and the second one - 1024 megabytes.

The aforementioned byte is the smallest memory unit consisting of 8 bits. They contain one of two values that are usually specified interchangeably as:

1 (one)

0 (zero)

or

TRUE

FALSE

Thus, the bit is information coincident with the record in the binary system, which will be discussed later in the book.

Bits in memory are numbering in a specific way - ascending to the left. Often one can find the term "youngest" and "oldest" or "least significant" bit. Sometimes the term "upper" and "lower" bytes are also used, which means the same thing.

All these terms result from the fact that their change affects the number to a different degree. This is a consequence of the binary notation, which we write about in the next section.

To program 16- or 32-bit processors, such as the Motorola 680x0 series installed in the Amiga, we need to learn the concept of the so-called "word". There are:

- "word" - consisting of 16 bits (two bytes)
- "long words" that are 32-bit, or represent the equivalent of four bytes

The word can be any number in the following range:

0 - 65536

whereas a long word can be a much larger number, as below:

0 - 4294967295

Therefore, words can obtain values from -32768 to +32768, while one byte has a range:

0 - 255

so it can have a value between -127 and +127. It should be noted that individual data types do not only differ in values, but are also stored in the computer's memory in other ways. We will talk about the practical application of this information later.

The machine language does not use the decimal system, and instead we use hexadecimal (or hexadecimal) and binary number systems (ie binary system). That is why we will now tell more about this.

NUMBER SYSTEMS

- Hexadecimal system

In the hexadecimal system, the basis is the number 16, instead of 10 - as in the decimal system. Therefore, instead of writing subsequent numbers:

0 1 2 3 4 5 6 7 8 9

we use a set of numbers from 0 to 9 and letters from A to F, as below:

0 1 2 3 4 5 6 7 8 9 A B C D E F

They represent values from 0 to 15. In other words, the letters denote the following numbers in the decimal system:

A = 10

B = 11

C = 12

D = 13

E = 14

F = 15

Therefore, the number 10 written in the hexadecimal system means 16 written in decimal. Initially, it may seem strange that the number is determined using a letter, but over time, you can appreciate this system. You can save a lot of values in this way for a shorter time, and they can look more orderly.

For example, if we use numbers with a maximum value of 255 decimal - which gives a hexadecimal FF - everything can be entered as just two characters. For simplicity, look below:

<u>decimal</u>	<u>hexadecimal</u>
47	2F
156	A5
255	FF

Of course, the entry can be more complicated when we want to use larger values, but the principle is always the same.

- Binary system

The binary system is harder to understand. To write numbers, we use only two digits: 0 and 1. Here's an example:

110

This record means the number 6 written in decimal. How to calculate it? Take into account the successive powers of the number 2 in the columns counting from right to left. To better understand this, we break the above record into three separate columns:

<u>1</u>	<u>1</u>	<u>0</u>
2^2	2^1	2^0

In the upper part, we see a binary record after breaking the columns into numbers, while at the bottom - the corresponding decimal values. As you can see, the powers of the number 2 are counted from zero, and in each subsequent column the exponent increases by one.

After calculating the value it will look like this:

<u>1</u>	<u>1</u>	<u>0</u>
4	2	1

To get the final result, multiply values from individual columns and then add them together. We must therefore perform the following operation

$$1 \times 4 = 4$$

$$1 \times 2 = 2$$

$$0 \times 1 = 0$$

Now we add these values:

$$4 + 2 + 0 = 6$$

In this way we have obtained the final result. Number 110 written in binary means 6 written in decimal. Again - it will not be as easy when saving larger values, for example:

$$1010 = 10$$

$$1101001 = 105$$

$$100011011 = 283$$

Certainly, calculations will take a lot more time, but this way of writing numbers should be known, because we will often use it during programming.

Let us add that the powering in the program is written as the sign "^", i.e. for example:

$$2^4 = 2^4$$

$$2^1 = 2^1$$

We will also use this form later in the book.

We can distinguish many more numbering systems, but to begin with, you should master the above information. Otherwise, many of the operations discussed will be incomprehensible.

PROCESSOR REGISTERS

There is another type of memory that is connected to the processor this time. They are "registers", i.e. small memory cells placed inside the processor. They are used to store temporary data, such as calculation results. The processor performs operations using its internal registers by copying data from the memory and then transferring the results back into memory.

- Types of registers

There are several types of registers. 680x0 series processors have 8 so-called "data" registers and the same number of "address" registers. The first of these have the following designations:

D0 D1 D2 D3 D4 D5 D6 D7

According to their name, they are used to store data (numbers), for example the results of calculations. We can also perform math operations on them.

The "address" registers have other symbols:

A0 A1 A2 A3 A4 A5 A6 A7

They can be used in a very similar way, except that you can not perform instructions on "byte" data, but only "word" or "long word". They are used to store memory addresses, using them it is also possible to access memory by reading or writing data. Some instructions are intended for use only in relation to data registers.

The operating of registers is a complicated topic and we will deal with it in the further part of the book. Amiga also has registers related to specialized circuits that are mounted on the computer's motherboard. You can read them in the appendix at the end of the book. I will say how to use them later.

MULTITASKING

Let's add a few words about the operation of the Amiga operating system, because in the following chapters we will also talk about the use of system components. The AmigaOS kernel is characterized by multitasking with expropriation. It is based on the presence of so-called scheduler, i.e. an algorithm that distributes processor time and access to computer resources between individual working tasks. As a result, many programs may seem to be running simultaneously, because you do not have to switch off one to start another one.

Multitasking on the Amiga does not require the use of very fast computers, this mechanism can be used even on simple hardware configurations. The work of many programs in parallel does not significantly reduce the speed of the operating system.

The system also allows the coupling of functions of many programs to get the automation of activities or using them in an unusual way. You can combine specific functions of different programs in such a way that they automatically perform functions that require a lot of work. Two seemingly identical programs can work and look completely different to different users.

These possibilities are created by the ARexx scripting language and the so-called system clipboard. ARexx support requires learning programming elements, although you can do without it by properly using ready-made programs.

Many of them also have options that allow you to use the scripting language directly from the window or program menu.

The above information should be taken into account when writing your own program. We can, of course, disregard the recommendations of the system creators, as well as the habits of users, but it will certainly affect the popularity of the finished program.

Also note that a lot depends on the purpose of the created algorithms. If you plan to run the program on unbuilt Amiga models, you will have to look for solutions that use a small amount of memory and operate relatively quickly. When you plan to direct your program to users with extensive Amiga configurations, you will not have to look for acceleration in every possible place.

Now let's get to the basic thing, which is to run the editor where you will be introducing your program. One of them is the popular "Asm-One", which was released in many versions also operating on the usual Amiga 500 with 1 megabyte of RAM, without a hard disk.

EDITOR

STARTING FROM THE FLOPPY DISK

To use the capabilities of a machine language, we must run the appropriate editor, through which we will enter instructions, test their operation and perform many other necessary operations. As we have already mentioned, one of such products is "Asm-One", which can be found in versions operating on every Amiga, even the simplest models equipped with 512 kilobytes of memory. The individual releases are not fundamentally different, although they may have slightly different function names or the placement of buttons in windows.

The most convenient solution is to install a program on your hard drive, but if you have an Amiga without a disk controller, you need to obtain a version that only works with a floppy disk. On the Internet you can find a lot of files in ADF or DMS format, which you just need to burn to a floppy disk and then start the Amiga.

The boot disk boot sequence may be different, but after running "Asm-One" you should get a screen similar to the following:

ASM-One V1.48 By T.F.A. Source 0 »

--- ASM-One V1.48 Motorola 68k/PPC Macro Assembler (KS 2.x/3.x) ---

Original 68k Coding by Rune Gram-Madsen (1990-1991)

Additional Coding and PPC Coding by T.F.A. (1991-2002)

Public Release, 31-12-2002 (Rev.482)

Usually, you will find quite old versions of "Asm-One" on ready-made disks. You can work on them, but it is also possible to download a newer version from the well-known Aminet.net website. To prepare the program for work Your Amiga must have at least 2 megabytes of memory. Otherwise, there will be insufficient memory space to save all necessary files.

You may have to limit yourself to a ready, older version, but below we will present a solution for users of slightly more complex configurations who do not have a hard drive. This can be, for example, the Amiga 500 or 2000 with additional memory, which requires a hard-to-access controller to be equipped with a drive.

For a start, a small reminder. Remember that you can perform all operations by loading the original Workbench floppy disk, which starts the minimal configuration, taking up as little memory as possible. In addition, you can perform a "no star sequence" boot, thus calling the so-called "Boot Menu" and selecting the option:

Boot With No Startup-Sequence

or by going to "Advanced Options ..." and then changing the "Startup-Sequence" function from "ENABLED" to:

DISABLED

It all depends on the version of Kickstart in your Amiga. The first option applies to version 3.0 and 3.1, while the second option - version 2.0.

Please note that in the case of Kickstart 1.2 or 1.3, the "Boot Menu" function is not available and in this case - after starting the Workbench - you will have a slightly smaller amount of memory available.

After these technical comments, we can now proceed to download files from the Internet. Go to page:

aminet.net

and enter the following phrase in the search box on the right ("Search"):

Asm-One

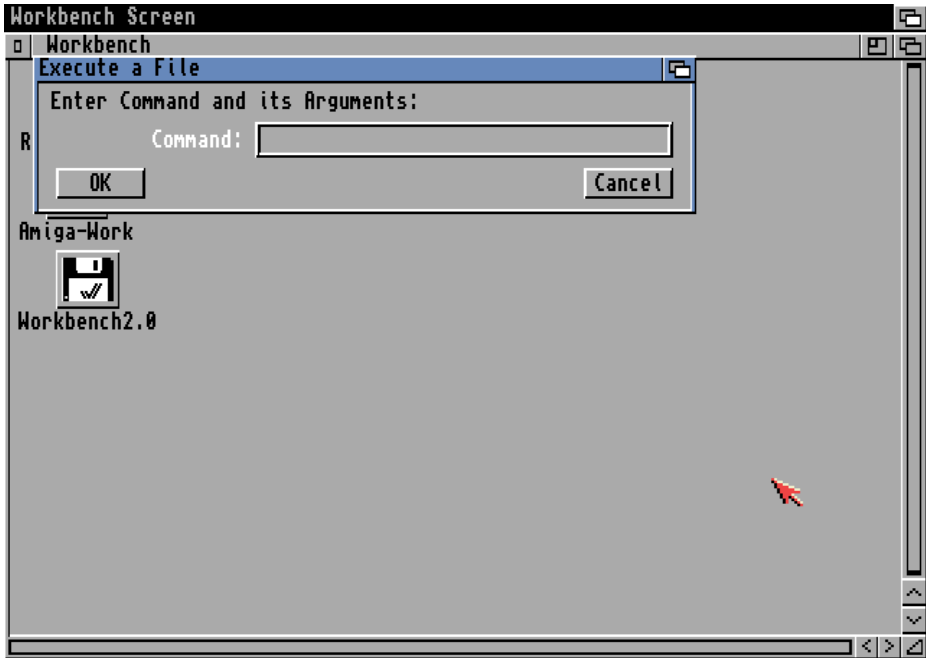
After a while, the list will show one item named "ASM-One.lha". It should be downloaded and then unpacked using the LHA archiver. You can also download it from the Aminet website, but enter the following text in the search box:

lha.run

Only one item will appear on the list again. Save the file to disk, preferably in the same location as the previous archive. You can use "Ram Disk" or an empty floppy disk for this purpose. Both files will fit nicely, because together they occupy only about 400 kilobytes.

Now go to the "Shell" window on Workbench and change the current directory to the place where you saved the files. You can do this by selecting the option "Execute Command" from the top menu called "Workbench".

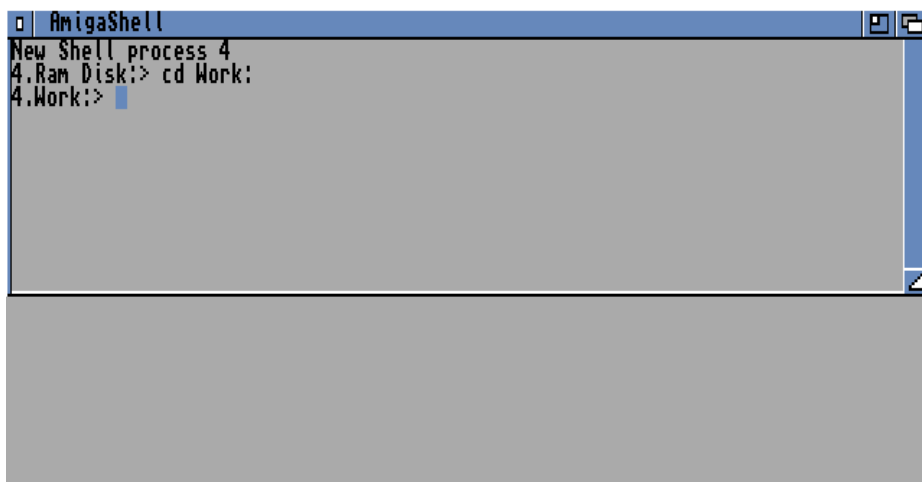
A small window will appear to enter command names that do not give access to the AmigaDOS window by default. It should look like this:



Of course, on the left you can see disk icons that may look different to you or have different names. Now just enter the NEWSHELL command in the text box, followed by the CD and the disk name where you saved the files.

The symbol should be ended with a colon character, because such rules are valid when entering the names of devices (also disks) within the AmigaDOS windows, ie "CLI" and "Shell".

Performing the operation confirm with the ENTER key. As a result, you should get the effect:



If you only use one floppy disk drive, you will have to change the media repeatedly during work. Unfortunately, with a limited amount of memory, there is no other way out, however, note that you will only have to perform these operations once.

Of course, your directories may have different names than in the picture, so you'll see a different one in front of the cursor. However, it must always point to the directory where you saved the previously downloaded files. To be sure, you can display the contents of the directory using the DIR command. Just enter its name and symbol:

#? .run

Thanks to this you will see only files ending in the extension ".run" in the window, so one of our files will appear. After pressing ENTER the information should look like this:



With us on the disk there are two items with this extension, but now we will be interested in the file marked with a frame. It does not matter if you have more items in the catalog, but there must be a file with the LHA program in it.

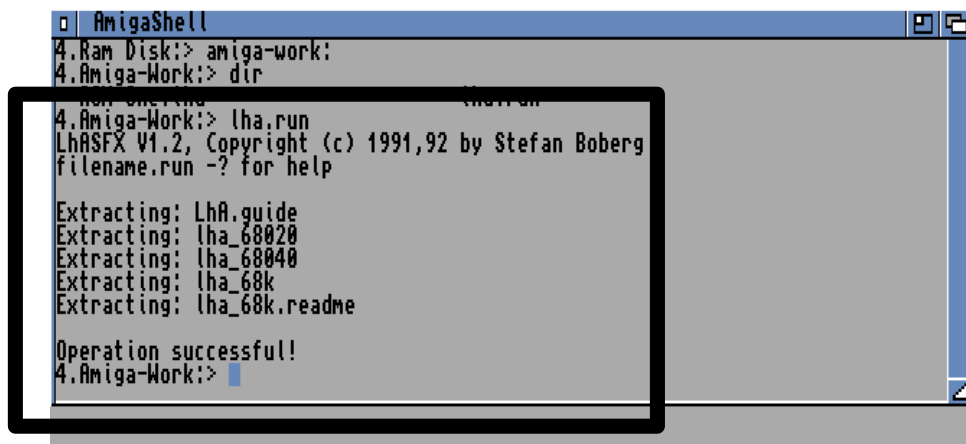
When we are sure that we have access to the files, we can proceed with further work. In the "Shell" window, enter:

lha.run

and press ENTER. After a while new messages will appear and most of the lines should start with the name "LHA". The first part will be information about the version of the program and its author, while the next one with the inscription:

Extracting:

they point to decompressed files. As in the picture:

A screenshot of an AmigaShell window titled "AmigaShell". The window shows a command prompt at "4.Ram Disk:> amiga-work:". The user enters "4.Amiga-Work:> dir", which lists files including "lha.run". The user then enters "4.Amiga-Work:> lha.run". The output shows "LhASFx V1.2, Copyright (c) 1991,92 by Stefan Boberg" and "filename.run -? for help". A list of files to be extracted is shown: "Extracting: LhA.guide", "Extracting: lha_68020", "Extracting: lha_68040", "Extracting: lha_68k", and "Extracting: lha_68k.readme". The final output is "Operation successful!" followed by a new prompt "4.Amiga-Work:>". A black rectangular box highlights the entire output section from "Extracting: LhA.guide" to the new prompt.

```
AmigaShell
4.Ram Disk:> amiga-work:
4.Amiga-Work:> dir
4.Amiga-Work:> lha.run
LhASFx V1.2, Copyright (c) 1991,92 by Stefan Boberg
filename.run -? for help

Extracting: LhA.guide
Extracting: lha_68020
Extracting: lha_68040
Extracting: lha_68k
Extracting: lha_68k.readme

Operation successful!
4.Amiga-Work:>
```

As you can see, the last line before the new prompt (where the cursor is) should contain the following inscription:

Operation successful

If it is different, it means that the file has not been saved correctly and has lost the so-called AmigaDOS attributes. Thanks to them, the file has certain features, among others using them the system determines whether a given file can be unpacked by entering its name - as in this case. To fix the problem you have to enter one more line, this time with the PROTECT command. You should type:

protect lha.run +rwed

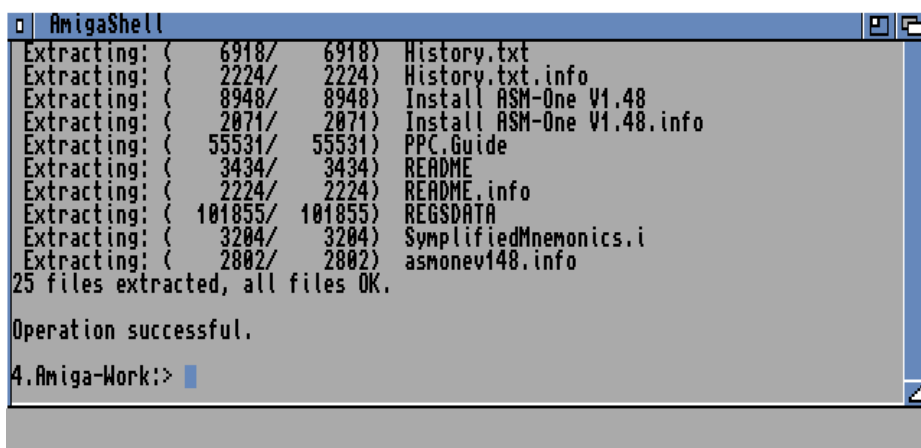
Now the file will get the appropriate attributes and you will be able to use it. If everything went well you should still change the name of one of the files. To do this, enter another line:

```
rename lha_68k lha
```

This operation is not necessary, but it will help to unpack the archive with the "Asm-One" editor. After pressing the ENTER key, the cursor should go to the next line without displaying additional information. It means everything is fine. In this way you have obtained the LHA program ready to work. To unpack the archive, use the "Disk Framework", because the whole thing takes about 900 kilobytes and will not fit on a floppy disk. Therefore, the next step should be the line:

```
lha x Asm-One.lha RAM:
```

which tells you to unpack the file into memory. The operation will take longer, and during the execution, the names of subsequent files and their volumes will appear in the window. As below:



```
AmigaShell
Extracting: ( 6918/ 6918) History.txt
Extracting: ( 2224/ 2224) History.txt.info
Extracting: ( 8948/ 8948) Install ASM-One V1.48
Extracting: ( 2071/ 2071) Install ASM-One V1.48.info
Extracting: ( 55531/ 55531) PPC.Guide
Extracting: ( 3434/ 3434) README
Extracting: ( 2224/ 2224) README.info
Extracting: ( 101855/ 101855) REGSDATA
Extracting: ( 3204/ 3204) SymplifiedMnemonics.i
Extracting: ( 2802/ 2802) asmonev148.info
25 files extracted, all files OK.
Operation successful.
4.Amiga-Work:>
```

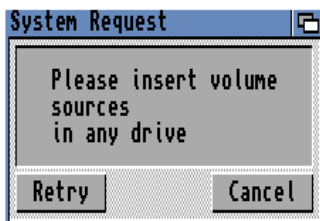
When everything is done correctly, the following line should be visible:

Operation successfull.

If this is the case, it means that all files have been saved and you can proceed to further work. Enter the following line:

assign Sources: RAM:

Thanks to it, we will create a logical device called "Sources:", which the editor requires, because it is the default directory when calling disk operations. Using the ASSIGN command is not absolutely necessary, but if you omit it, the program will display windows similar to the following:



All you need to do is to click the "Cancel" button to make everything work properly, but it's not very convenient. That's why we suggest creating a logic device.

Now we can start "Asm-One". You can do this by double clicking on the Workbench icon labeled "Asm-One_V1.48" or by entering the same name in the AmigaDOS window. In order not to reduce the amount of free memory, we will use the "Shell" window. Enter:

Asm-One_V1.48

and press ENTER. After a moment, the editor screen will appear on the screen:

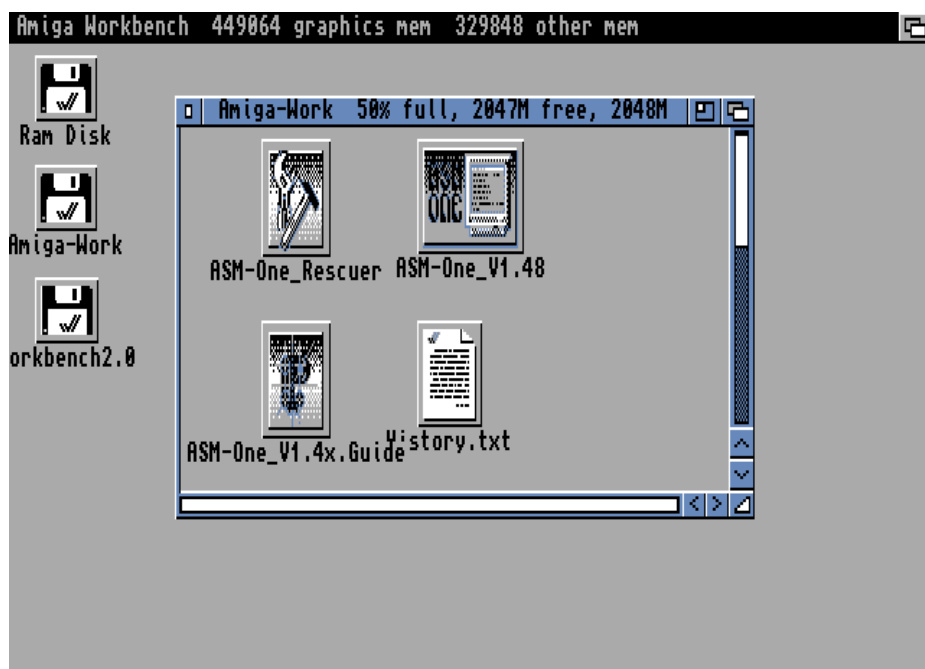
ASM-One V1.48 By T.F.A. Source 0 »

```
--- ASM-One V1.48 Motorola 68k/PPC Macro Assembler (KS 2.x/3.x) ---  
Original 68k Coding by Rune Gram-Madsen      (1990-1991)  
Additional Coding and PPC Coding by T.F.A.    (1991-2002)  
Public Release, 31-12-2002 (Rev.482)
```

From now on we start the right job using the machine language, because here we will run all Assembler features.

INSTALLATION ON HARD DISK DRIVE

Using floppy disks is not very convenient, so it is best to install "Asm-One" on your hard drive. It does not require many additional activities than those discussed in the previous section. However, after unpacking the archive, you should go to the "Ram Disk" window on the Workbench and run the installation script. After doing a double click on the icon, you will see a window containing several icons, as below:



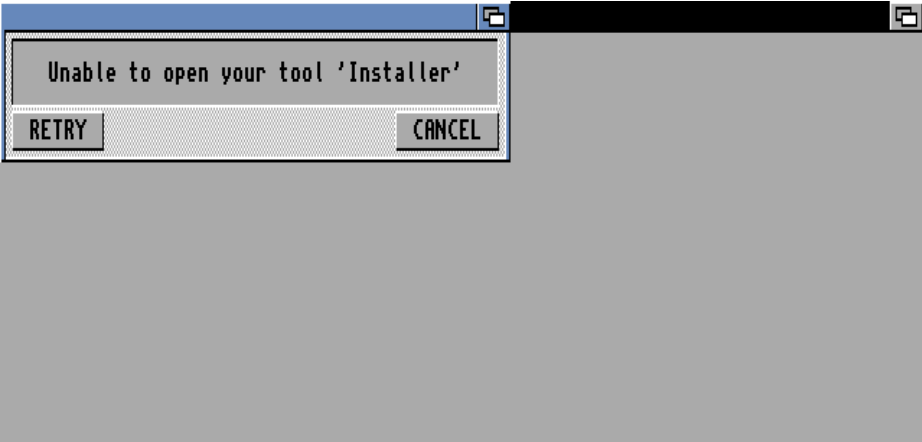
Scroll the contents of the window and double-click on the icon signed as:

Install ASM-One V1.48

You will launch a short installation program. Look how it should look like:



If you see another window, you probably do not have the INSTALLER command installed. In this situation, another window will appear on the screen:



To solve this problem, use the "Installer" available on Aminet. It is enough, just as before, to write a phrase:

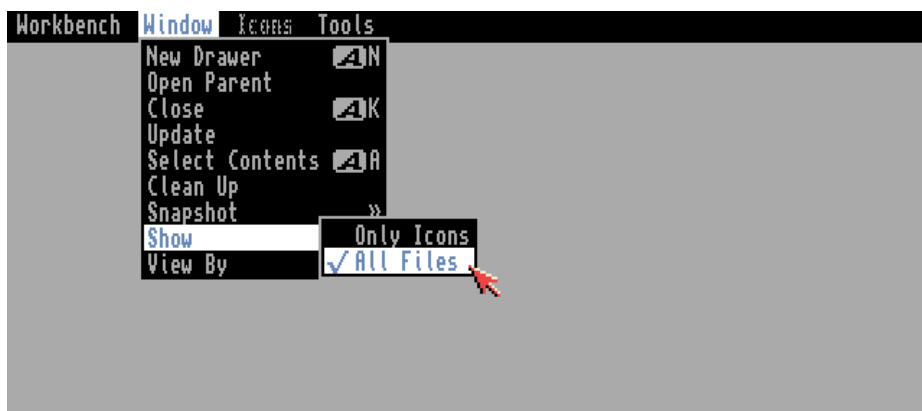
Installer 43

in the search box on the right. You will see two items on the list again. Download the file named "Installer-43_3.lha".

The installation method is not complicated. At the beginning of the archive, we unpack using the "Shell" window. Enter the line according to the following:

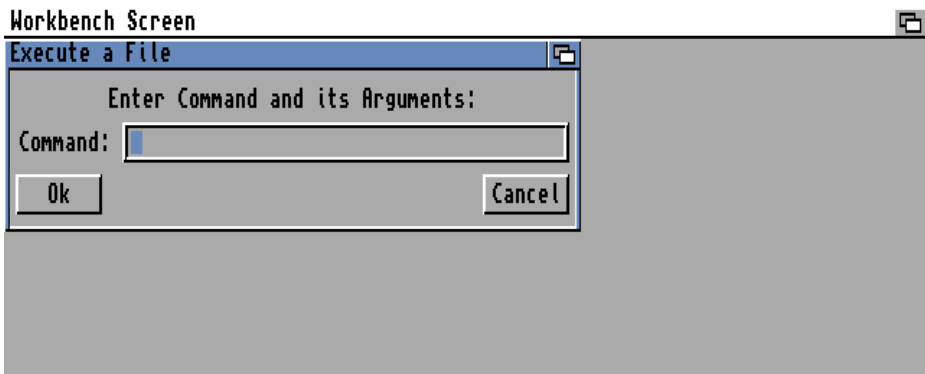
lha x Installer-43_3.lha RAM:

Of course, the files will be in the "Disk Disk". Go there and double-click on the "Installer43_3" icon. When the content is read, turn on the option to display all files, ie call up the top menu and use the "All Files" option in the top menu "Window" under the item "View ..." (Show.) For simplicity look at the illustration:



Now the icons will appear in the window. Find the item signed as "Installer", hover over it and press twice the left mouse button again. You will again see the "Execute Command" window.

Depending on the version of the operating system, it may look a bit different, but its function does not change.



Get the following entry:

copy Installer C:

and press ENTER. The file will be copied to the system directory "C". When everything is done correctly, no additional information will appear on the screen.

Then you can restart the "Asm-One" installation program. The first two messages are purely informative, so in both cases select the "Proceed" box. Then you can meet the information about the "ReqTools" package, which is required to run the editor. The selection windows will be created using the mentioned package.

If your library does not have a library with this name, you'll see new information:



In this situation, the installation will be interrupted, therefore select the "Proceed" button. When the window disappears, you must complete the missing files. Go back to the Aminet website and search for the phrase "ReqToolsUsr". The file will appear in the list:

ReqToolsUsr.lha

which should be downloaded to the disk, just as before. Call the "Shell" window again and change the current directory to the one where the file was saved. We have to unpack the archive, so enter the following line:

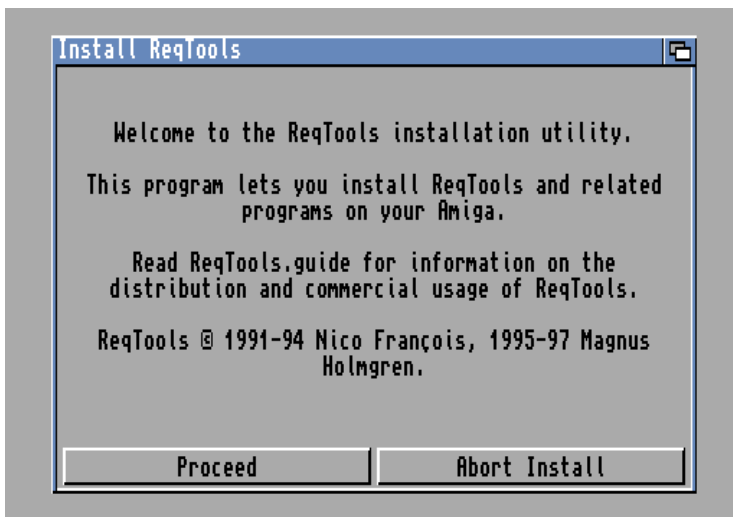
```
lha x ReqToolsUsr.lha RAM:
```

to save all files in "Disk Framework". Of course, instead of "RAM:" we can enter the name of the hard disk partition, but the temporary data will successfully fit in the memory, because they only take about 380 kilobytes.

Then on Workbench, go to the directory, where the archive was unpacked. There will be a catalog called "ReqTools". After reading its contents, several icons will appear on the screen. Hover over the icon:

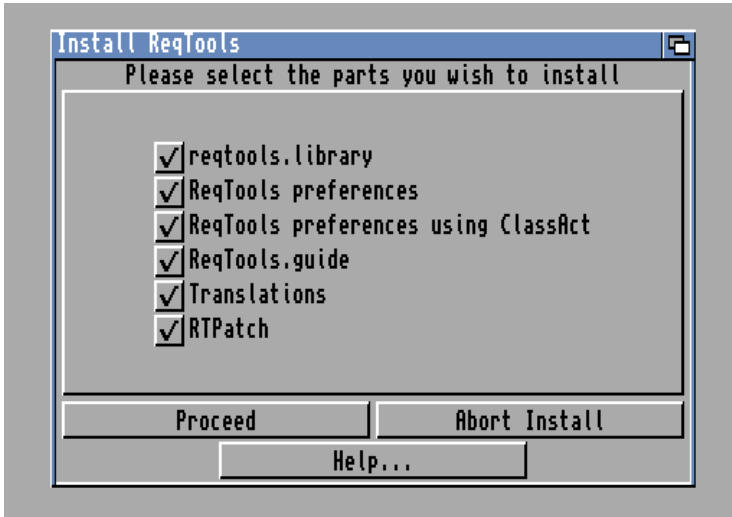
Install ReqTools

and press the left mouse button twice. You will launch the standard library installation program:



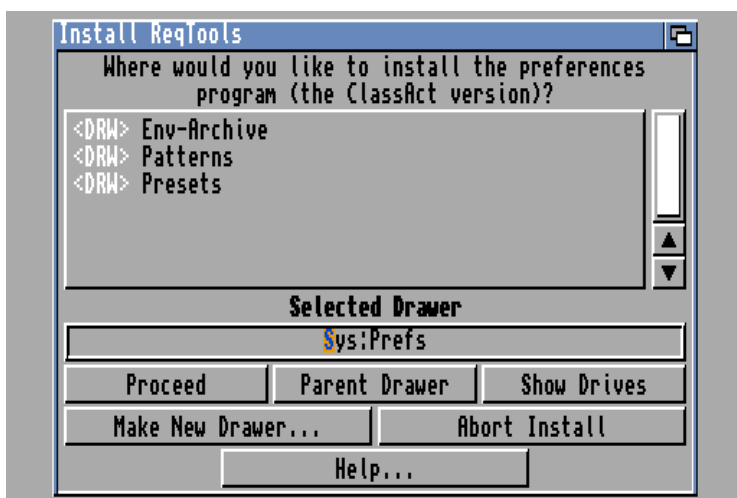
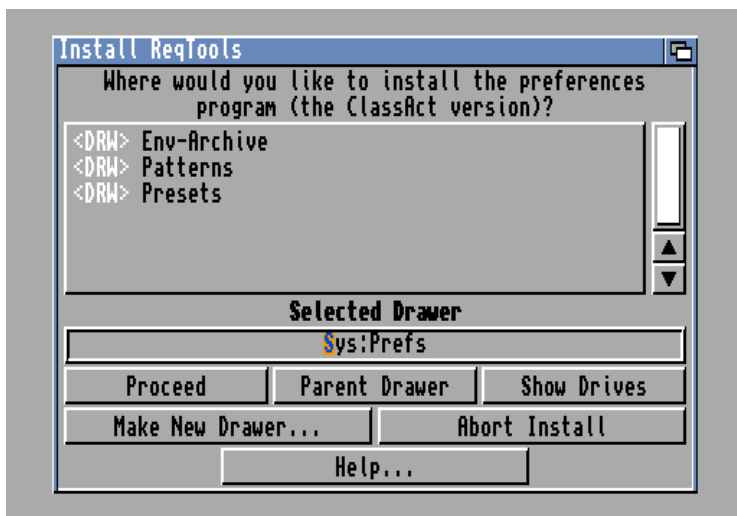
The messages are purely informative. Select the "Proceed" button, then "Proceed With Install" and then again "Proceed".

The files to be copied to the disk will be displayed:



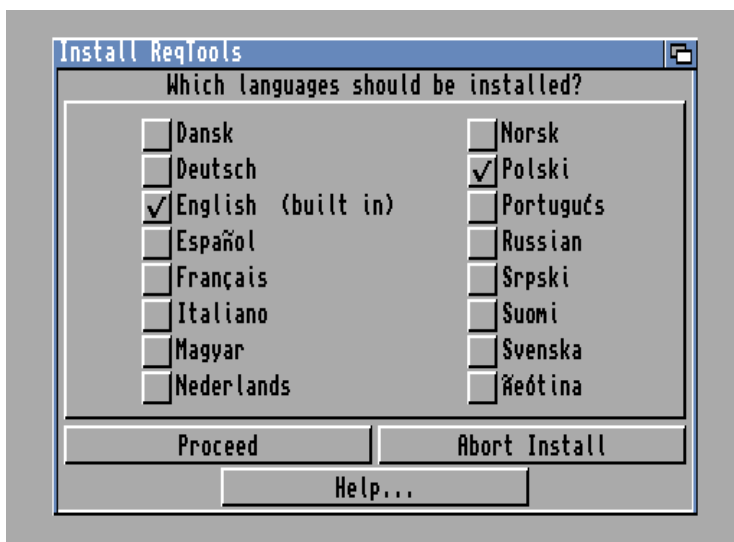
We have here the library itself, the first file with the extension ".library". In addition, you will install a program to change settings, documentation, as well as a program called "RTPatch", which will make all the selection windows used on the Workbench will use "ReqTools".

Do not change the settings of individual fields, but only use the "Continue" button once again. In the next two steps, the program will ask about the catalogs in which the program should be stored - the preferential program of the library and documentation. It will look like the following illustrations:



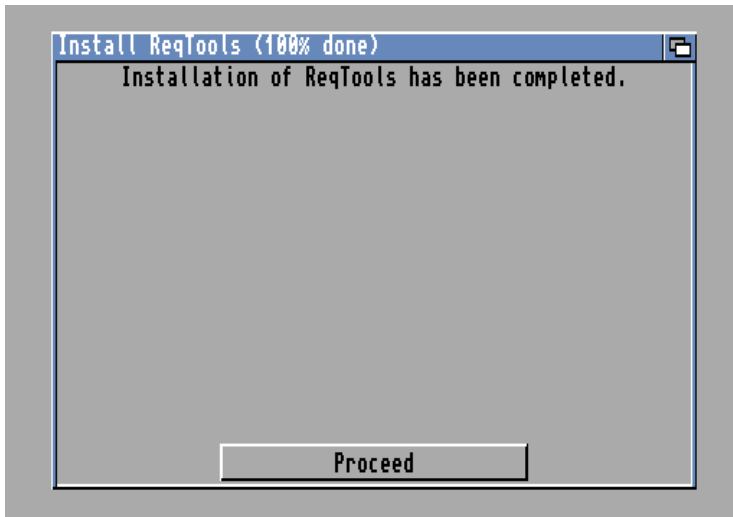
In both cases, do not change the visible access paths. Select only the "Continue" button twice.

Now you will see the language selection options that will be used by "ReqTools". Hover over the field next to the name of your language and press - this time only once - the left key. The button should be characteristically circled, just like the box next to the "English (built in)" option:



Then, once again use the "Proceed" button. The installer will ask for a directory in which the "RTPatch" program will be saved. As we mentioned, this is an add-on so that all system selection windows will be automatically displayed using the "ReqTools" library.

Select the button marked "Proceed" again. After a while, the installation of all necessary files on the disk will start. After the operation you will see the last message



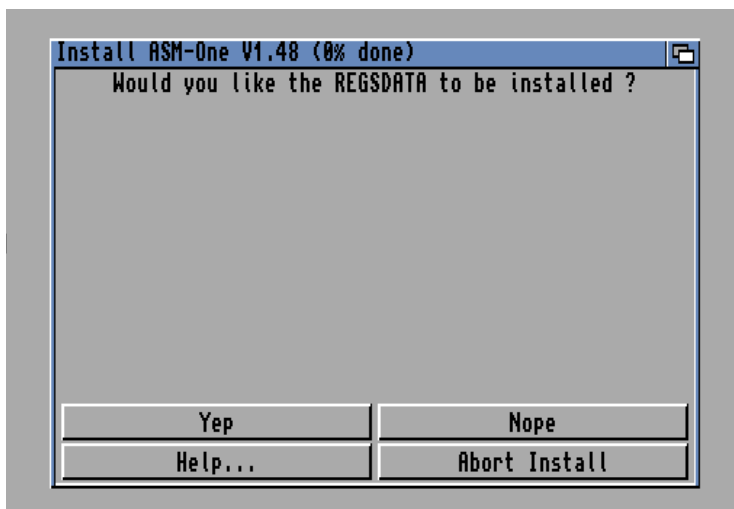
It's almost everything. Choose "Continue" and the window will close. The library has been correctly installed in the system. Now we can return to the "Asm-One" installation.

Start another installation program using the following icon:

Asm-One_V1.48

In the new window, do not change any settings, just double-click "Continue" to skip the initial information about the program and the possibility of installation "for trial".

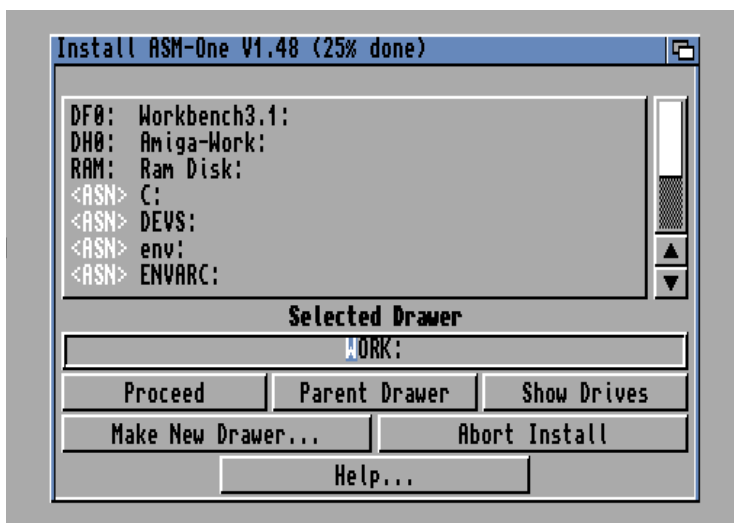
Now the error will not appear, but the question is whether you want to install the file named "REGSDATA":



Select the "Yep" box on the left. Three more times similar information will appear, but with different file names. Choose the "Sure" and "Absolutely" buttons - all will be displayed in the same place.

The program will ask you for the directory in which our editor will be installed. The window will change the appearance to a typical module, in which the names of available disks appear by default or the contents of the directory selected in the box below the list. Below you can see the buttons with which you can navigate the contents of the disk, as well as create new directories.

Using the list at the top of the window, you can choose any place on the disk, where there is free space. If you want to create an additional directory, select the field marked as "Make New Drawer ...". The whole window should look like this:



When you select a catalog, use the "Proceed" field. The copying operation will start and you must wait for it to finish. For a dozen or so seconds you will see more file names and a so-called progress bar showing the speed of copying data. Finally, the following inscription will appear:

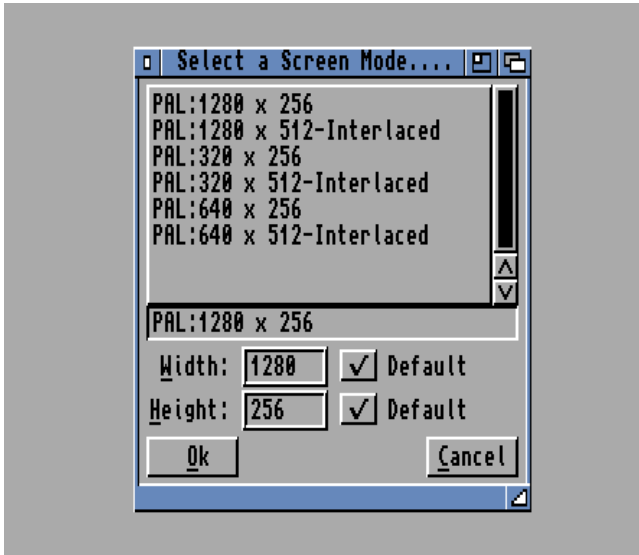
Thanks for using ASM-One V1.48

and at the bottom of the window, the familiar 'Continue' field. Select them twice and the installer window will disappear. This means that the editor has been installed and you can start it.

As before, if there is information about the lack of a volume named "Sources:", you can use the ASSIGN command to eliminate them. How to do this you can read in the previous section.

FIRST STEPS

After launching "Asm-One" a screen selection window will appear on the screen as below:



The message "Select a Screen Mode ..." means that you have to choose the display mode on which you want to work. "Asm-One" can work both on standard Amiga modes and using a graphics card.

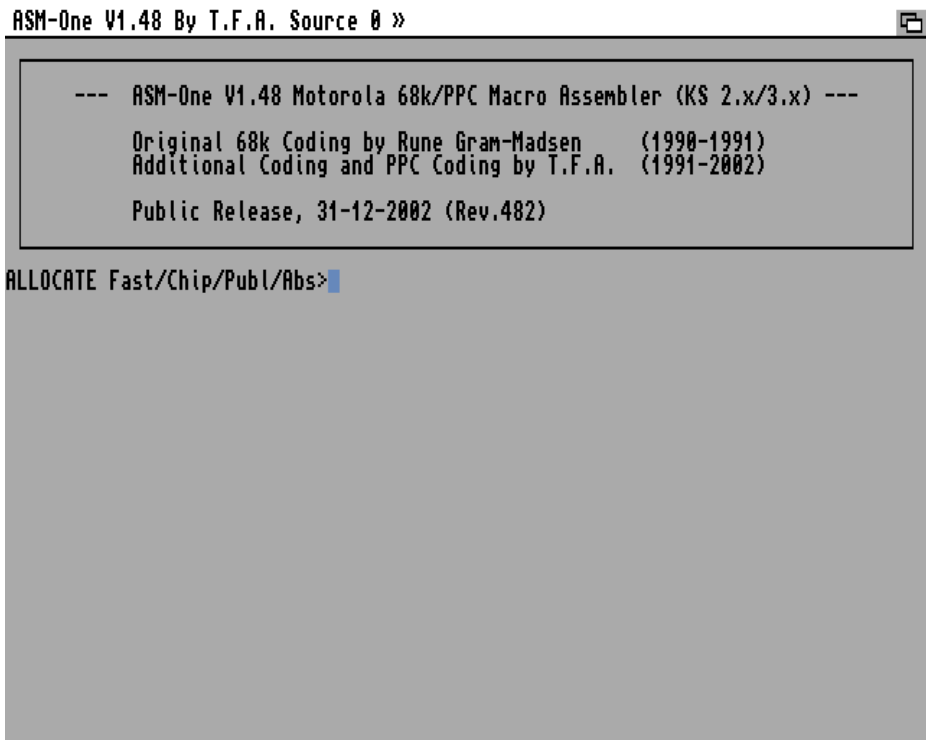
The list contains all currently available options, so if you only see the "PAL" modes - as in the above illustration - use the position:

PAL: 640 x 256

You will then get a typical Workbench resolution that does not take up much memory. At the same time, all the elements of the editor will be clearly visible.

You can also try several modes or run additional drivers on your system if, for example, you are using a VGA or Multisync monitor. After selecting one of the items on the list, use the "Ok" button at the bottom of the window.

Now you will see the main screen "Asm-One":



```
ASM-One V1.48 By T.F.A. Source 0 »  
--- ASM-One V1.48 Motorola 68k/PPC Macro Assembler (KS 2.x/3.x) ---  
Original 68k Coding by Rune Gram-Madsen (1990-1991)  
Additional Coding and PPC Coding by T.F.A. (1991-2002)  
Public Release, 31-12-2002 (Rev.482)  
ALLOCATE Fast/Chip/Publ/Abs>
```

Note the line containing the following message:

ALLOCATE Fast/Chip/Publ/Abs>

Here you can select the type of memory in which your program will be stored. Enter one of the four letters denoting:

- **F** - Fast type memory
- **C** - Chip memory

- **P** - "public" memory, ie the largest available block of memory

You can also enter one of the symbols separated by a slash sign, for example:

Chip

but it's more convenient to use a shortcut, i.e. only the first letter. Remember that the graphics and sound data to be used by Amiga specialized chips must be saved in the graphics memory (or Chip).

It is also good practice to store the program in public memory, which will usually use Fast memory, unless your Amiga is deprived of it. Then, of course, all operations will have to be carried out on the Chip type memory.

After selecting the place where the program is to be found, we need to indicate how large the block of memory should be allocated. Therefore, the next message will appear on the screen:

ASM-One V1.48 By T.F.A. Source 0 »

```
--- ASM-One V1.48 Motorola 68k/PPC Macro Assembler (KS 2.x/3.x) ---  
Original 68k Coding by Rune Gram-Madsen    (1990-1991)  
Additional Coding and PPC Coding by T.F.A.  (1991-2002)  
Public Release, 31-12-2002 (Rev.482)
```

```
ALLOCATE Fast/Chip/Publ/Abs>Chip  
WORKSPACE (Max.1888) KB>
```

After the words:

WORKSPACE (Max.1888) KB>

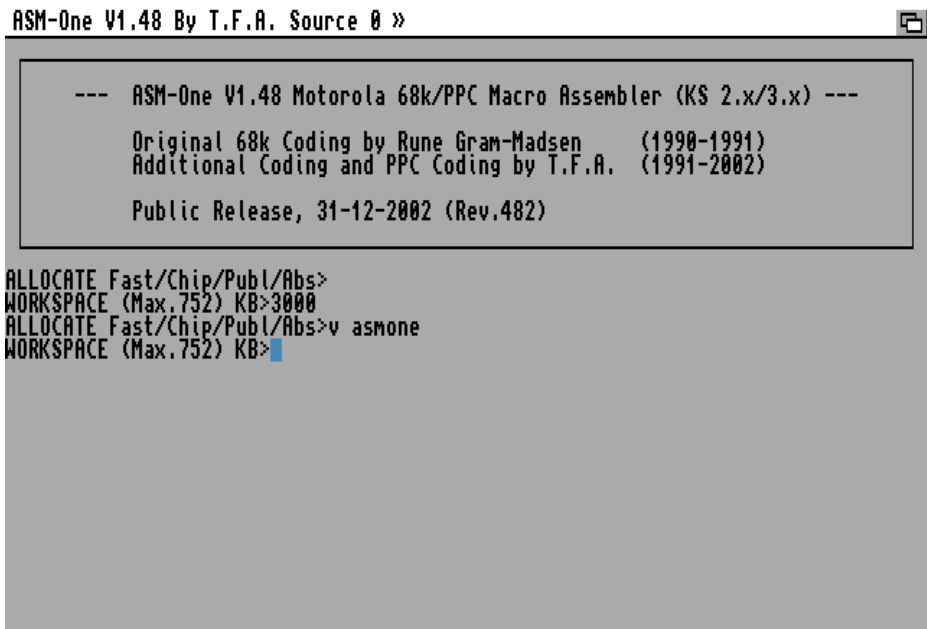
you should enter the size of memory that interests you. Of course, the number next to the word "Max" means the maximum size that we can choose and it will vary depending on the amount of memory installed in the computer.

The block allocated must be large enough to accommodate the program and the data it will use. If you are not sure how big the program will be, use the maximum size.

DEFAULT PARAMETERS

- Memory

If you run the program from the hard disk, it may happen that you will not be able to enter the memory parameters and they will be "filled" automatically. On the screen, the cursor will then automatically jump to the next line. It may look like this:

A screenshot of a DOS-style window titled "ASM-One V1.48 By T.F.A. Source 0 »". The window has a grey background and a black border. Inside, there is a text box with a black border containing the following text: --- ASM-One V1.48 Motorola 68k/PPC Macro Assembler (KS 2.x/3.x) ---
Original 68k Coding by Rune Gram-Madsen (1990-1991)
Additional Coding and PPC Coding by T.F.A. (1991-2002)
Public Release, 31-12-2002 (Rev.482)
Below the text box, the command prompt shows the following text: ALLOCATE Fast/Chip/Publ/Abs>
WORKSPACE (Max.752) KB>3000
ALLOCATE Fast/Chip/Publ/Abs>v asmone
WORKSPACE (Max.752) KB> The cursor is at the end of the last line.

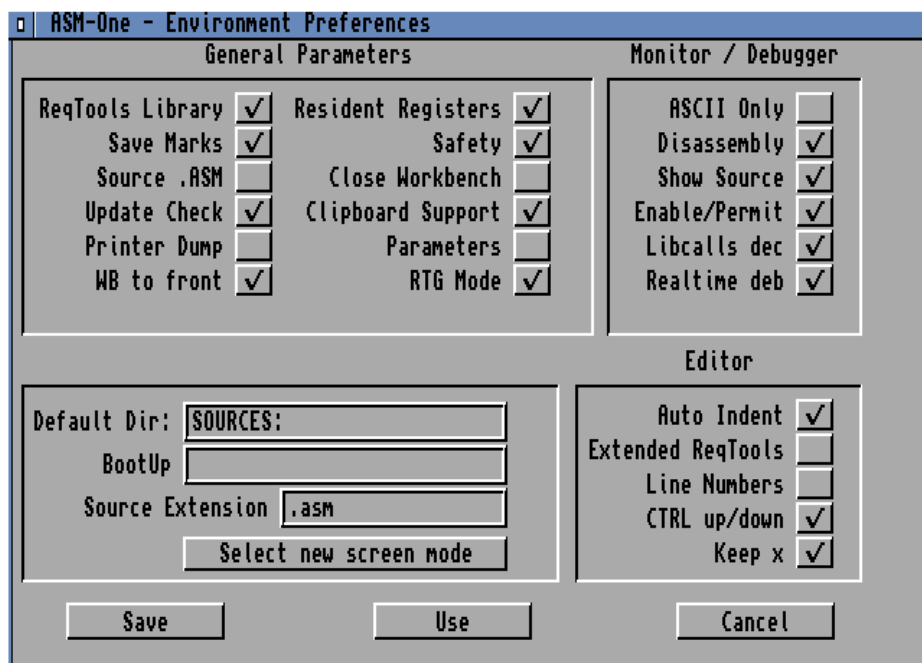
```
ASM-One V1.48 By T.F.A. Source 0 »  
  
--- ASM-One V1.48 Motorola 68k/PPC Macro Assembler (KS 2.x/3.x) ---  
Original 68k Coding by Rune Gram-Madsen (1990-1991)  
Additional Coding and PPC Coding by T.F.A. (1991-2002)  
Public Release, 31-12-2002 (Rev.482)  
  
ALLOCATE Fast/Chip/Publ/Abs>  
WORKSPACE (Max.752) KB>3000  
ALLOCATE Fast/Chip/Publ/Abs>v asmone  
WORKSPACE (Max.752) KB>
```

This is due to the fact that "Asm-One" is defending the definition of default work parameters, called after starting, regardless of the user. You can change this by using the "Environment" option, which is available in the top menu called "Assembler".

To get the mentioned function, select the item "Preferences":



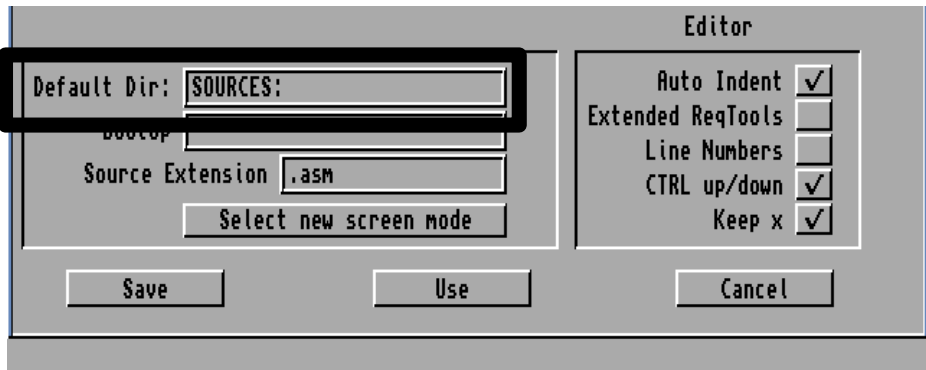
A large window will appear on the screen where you can change many different editor options:



The features executed automatically upon startup contain a text field marked as "BootUp". The default entry means that the program will react automatically and complete the entry as if it were performed by the user. If you want to change this, delete the entire entry in this field, and then select the "Save" button at the bottom. Once you start "Asm-One" again, you will be able to influence all starting work parameters.

- Working directory

Another important field in the preferences window is "Default Dir":



As you can see, it contains the "Sources:" logical device that we talked about earlier. You can enter any access path here, where you save files. Thanks to this program will use it by default for disk operations and you will not have to manually select the directory.

A field described as "Source Extension" is also available below. The standard entry should not be changed, because it causes the program to accept a file with the following extension:

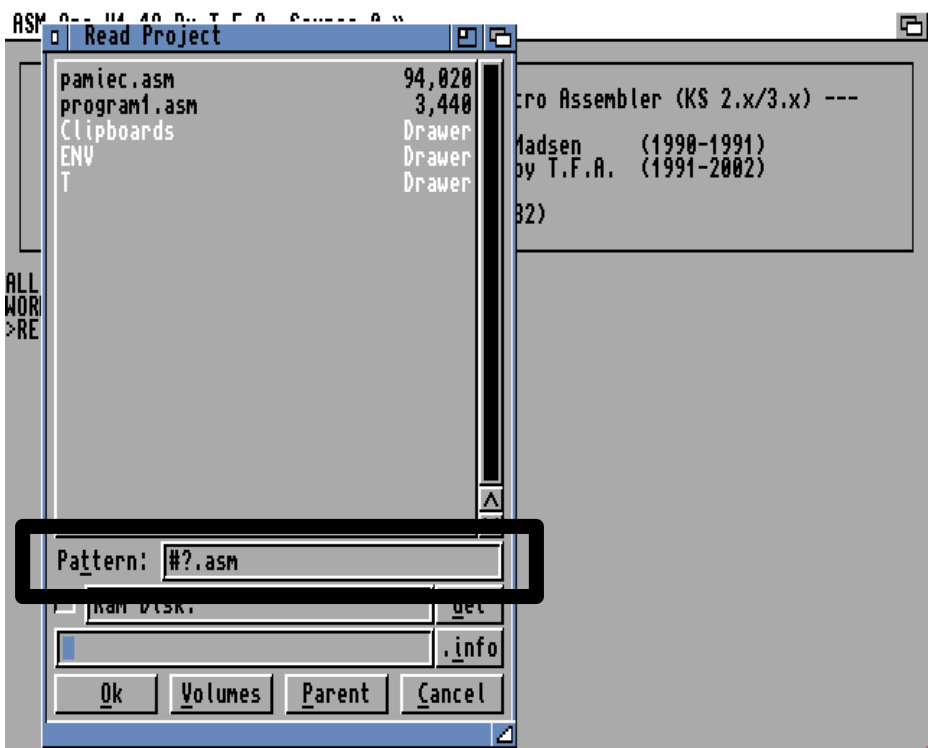
.ASM

as programs written in Assembler. In principle, it does not matter what naming we will use, but many files available on floppy disks or on the Internet will have such extensions. Therefore, it's best to change nothing.

Remember that in the selection windows you can apply the same principle so that only your programs are visible in directories. To make this happen in the "Pattern" field, use the following entry:

#?.asm

It should look like the next illustration:



The entry is the AmigaDOS filter, which is used throughout the entire operating system. This is a particularly convenient solution when you save many different files on one disk in one directory.

Let's add that the "#?" Character replaces any name or fragment of the name, regardless of the length. With it, you can specify the pattern of files that will be displayed, for example:

#?.info - for icons

or

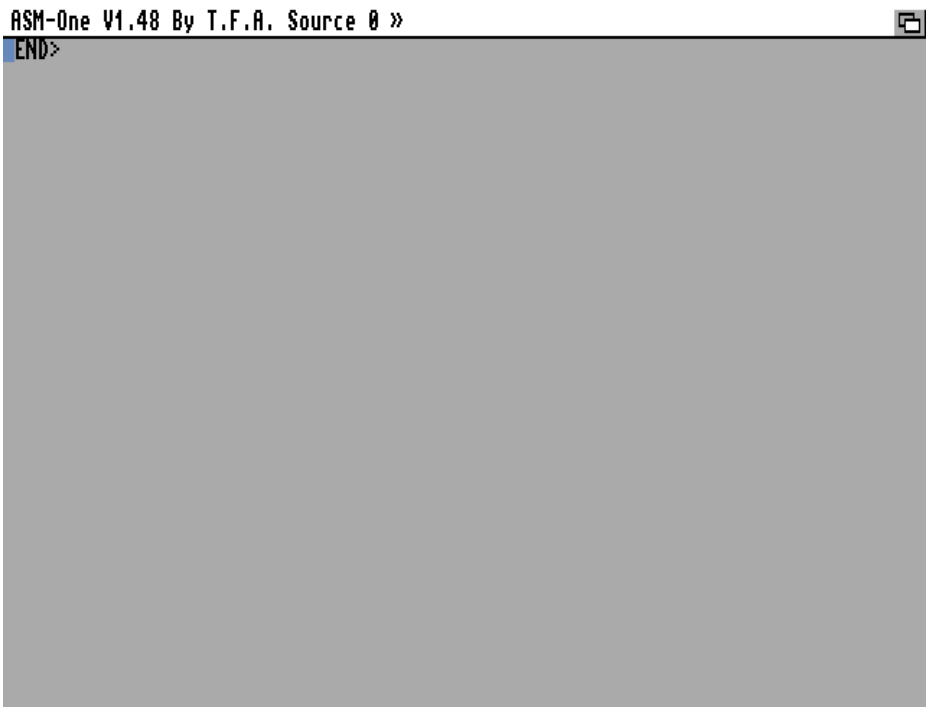
mod.#? - for music modules

The string "#?" Is characteristic for Amiga and is used in many programs during disk operations.

TYPING AND TESTING

THE PROGRAM

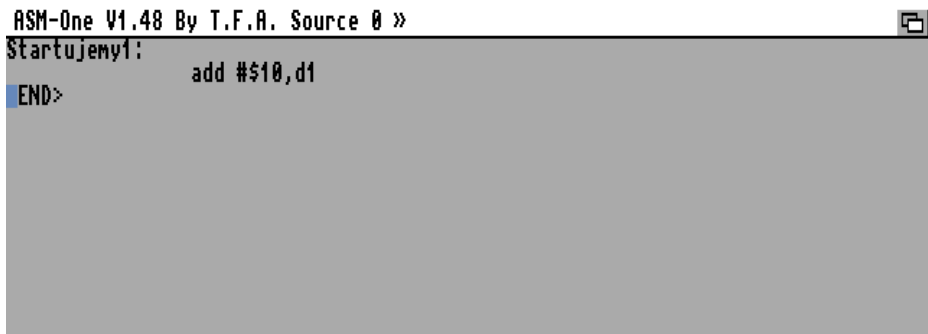
This is the simplest step in the editor, but requires a few explanations. "Asm-One" is started by default in the command line mode, which can be compared to the "Shell" window. Of course, we have other commands and functions, but the principle of operation is very similar. We can carry out the testing of the program here, as well as its launch, which we will talk about in a moment. First, however, you have to go to the screen where you will enter your program. To do this, select the "Editor" option from the top menu called "Assembler". The situation on the screen should look like this:



ASM-One V1.48 By T.F.A. Source 0 »

END>

The program is entered in the same way as in any other text editor, however, please note that it should start with a so-called label, i.e. a symbol with a colon character. For example, as in the next illustration:



```
ASM-One V1.48 By T.F.A. Source 0 »
Startujemy1:
    add #$10,d1
END>
```

Labels can and should also be used in other parts of the program when you want to go to a specific fragment or repeat the operation. You can read more about this topic in the sections entitled "Labels and" Loops ".

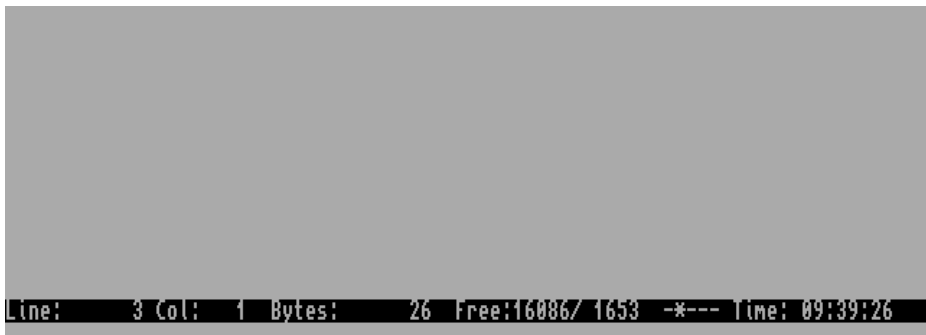
Another very important issue is the use of so-called "indents" in the program, i.e. spaces that separate labels from other commands. When you will not use them and enter the whole thing like this:



```
ASM-One V1.48 By T.F.A. Source 0 »
Startujemy1:
add #$10,d1
END>
```

The program will not be able to run. Indentations are often overlooked or treated with disrespect, in our case we must pay special attention to them. Inserting them does not require spending a lot of time, because you can do it comfortably with the Tab key.

Before we go any further, look at the status line at the very bottom of the screen. It contains a lot of interesting information about the program:



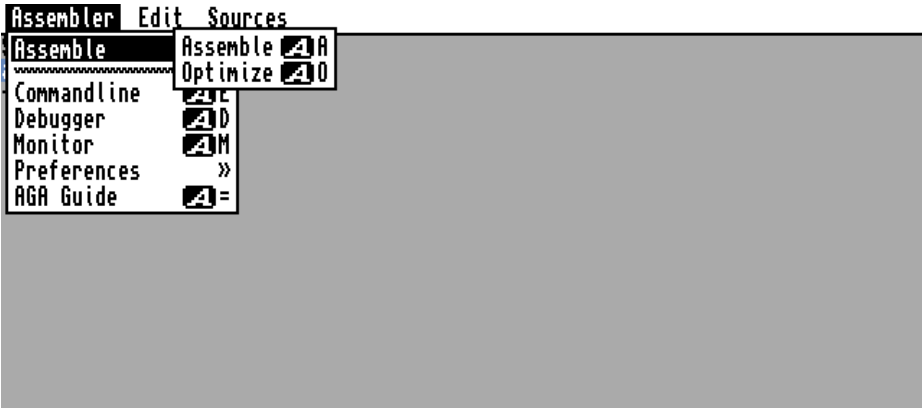
From the left you can see:

- **Line** - the current line in which the cursor is located
- **Col** - the current column in which the cursor is located
- **Bytes** - volume of the program entered (in bytes)
- **Free** - the amount of free memory
- **Time** - the current time set in the system.

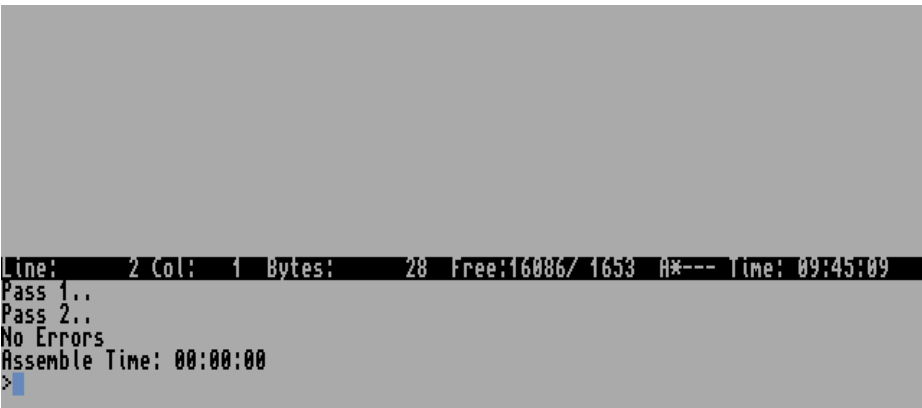
The status line may seem unnecessary to you, but its usefulness can be appreciated when writing larger programs.

When the program is ready, you can check whether it is correct from a technical point of view. Just select the "Assemble" option from the top menu

called "Assembler". Please note that this item develops two more functions, as in the following illustration:



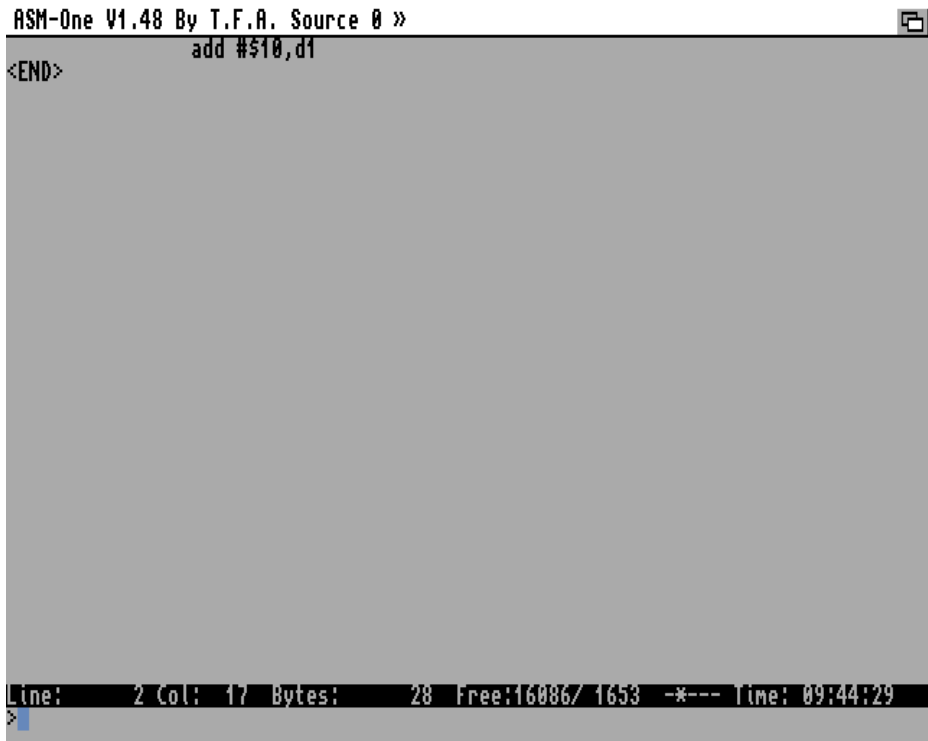
Now, at the bottom of the screen you should see the words "Pass1" and "Pass2":



The next line will show the words "No Errors", while we still have the specified time of the operation. Of course, all together means that no errors were found in the program.

When the "Assemble" function is called, the program content is translated into the machine's processor language. The operation is called "assembly" in Polish. Note, however, that this does not mean that the program will be launched right away.

The second option to test the code is to return to the command line using the "Exit" function from the top menu called "Edit" or "Edit funct.". You will not see the content entered in the editor, but there will be more space on the screen for other operations. It should look like this:



```
ASM-One V1.48 By T.F.A. Source 0 »
add #510,d1
<END>

Line: 2 Col: 17 Bytes: 28 Free:16086/1653 -*--- Time: 09:44:29
>
```

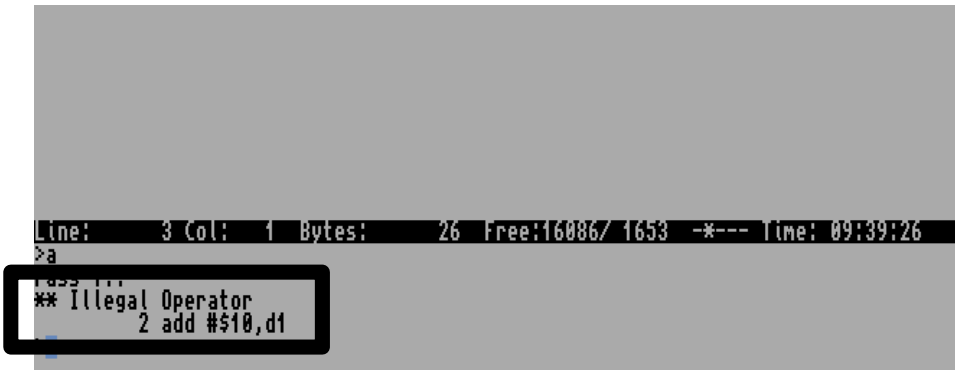
Now enter:

a

After pressing the ENTER key you should see the same messages as before, only now they will be displayed in a different part of the screen.

Of course you can again select the "Assemble" option from the top menu, but in this case using the command line is much faster and more convenient. On this occasion, notice that the menu options change depending on the module you are running. Slightly different functions appear in the editor and others appear on the command line.

Instead of the "No Error" message, you can also get error information, for example:

A screenshot of a debugger's command window. The status bar at the top shows 'Line: 3 Col: 1 Bytes: 26 Free:16086/1653 -*--- Time: 09:39:26'. Below it, the command line shows 'pa' followed by a prompt. The output area displays an error message: '** Illegal Operator' followed by '2 add #\$10,d1' on the next line. A black rectangular box highlights the error message and the line it refers to.

```
Line: 3 Col: 1 Bytes: 26 Free:16086/1653 -*--- Time: 09:39:26
pa
** Illegal Operator
2 add #$10,d1
```

How to solve these types of problems I will talk later. For now, it should be noted that the error message starts with two asterisks "**", and the next line indicates a problematic line. This does not mean that the editor suggests what should be changed. In addition, solving the problem will not always involve changing this line

When the program is correct, you can start it with another short command:

j

Please note that the running program may appear to be a system failure or cause other unexpected results. To try to stop it, press the SPACEBAR.

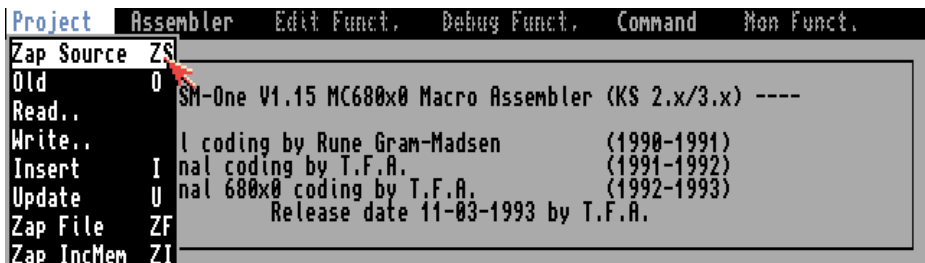
Remember that this does not mean that the program will fulfill the assumed functions. Programming in Assembler is difficult, because you operate directly on the computer's memory cells and, above all, the processor.

In many cases, a valid program can cause system hangs or other unforeseen reactions. In high-level languages, as in Amos or other Basica dialect, various types of errors are usually displayed. In "Asm-One" the computer can "hang" and the only advice is to restart the system or the editor itself.

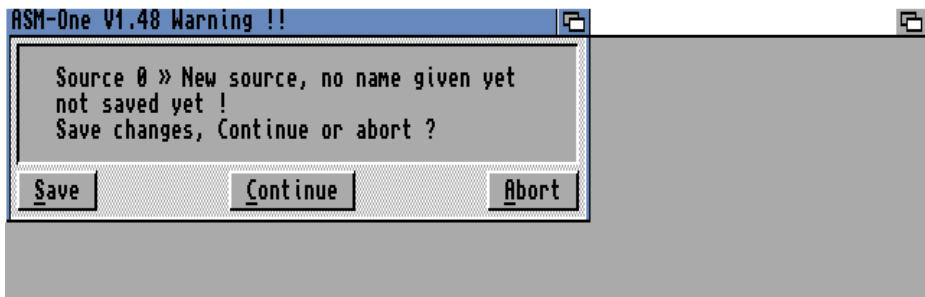
That's why it's so important to make working copies of the program, as we'll see in the next section.

REMOVING THE PROGRAM

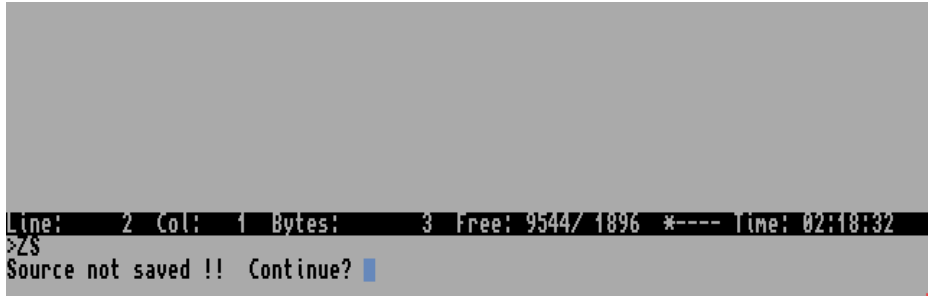
If you enter a program and then you want to delete it without leaving the editor, you should usually find an option similar to "New" or "New". The menu in "Asm-One" is unusual and this function is performed by the "Zap Source" function, which you will find in the menu named "Project":



After selecting it, you will be asked if you want to keep the current program. A window may be displayed at the top of the screen:



If you want to save the program in the editor, select the "Save" button. When it is your intention to delete the content, indicate the field in the middle, or "Continue". If you do not have the "ReqTools" library installed, you will see the following message on the command line instead of the window:

A screenshot of a terminal window with a dark background. The status bar at the bottom displays the text: "Line: 2 Col: 1 Bytes: 3 Free: 9544/1896 *---- Time: 02:18:32". Below the status bar, the prompt ">ZS" is visible. The main area of the terminal shows the message "Source not saved !! Continue?" followed by a blue cursor block.

```
Line: 2 Col: 1 Bytes: 3 Free: 9544/1896 *---- Time: 02:18:32
>ZS
Source not saved !! Continue? █
```

In this situation, press Y to confirm the operation. Remember that no other operation will be called automatically, so you will not see the option to load a new program. The "Zap Source" function simply deletes the content of the editor, thanks to which you gain more free memory for use.

DISK OPERATIONS

In most programs the functions related to reading and writing files are available in the top menu. It is no different in the case of "Asm-One", but we will not be able to use them when entering the program. You can notice this by calling the menu. First you have to go to the aforementioned command line using the "Commandline" option available in the "Assembler" pull-down menu.

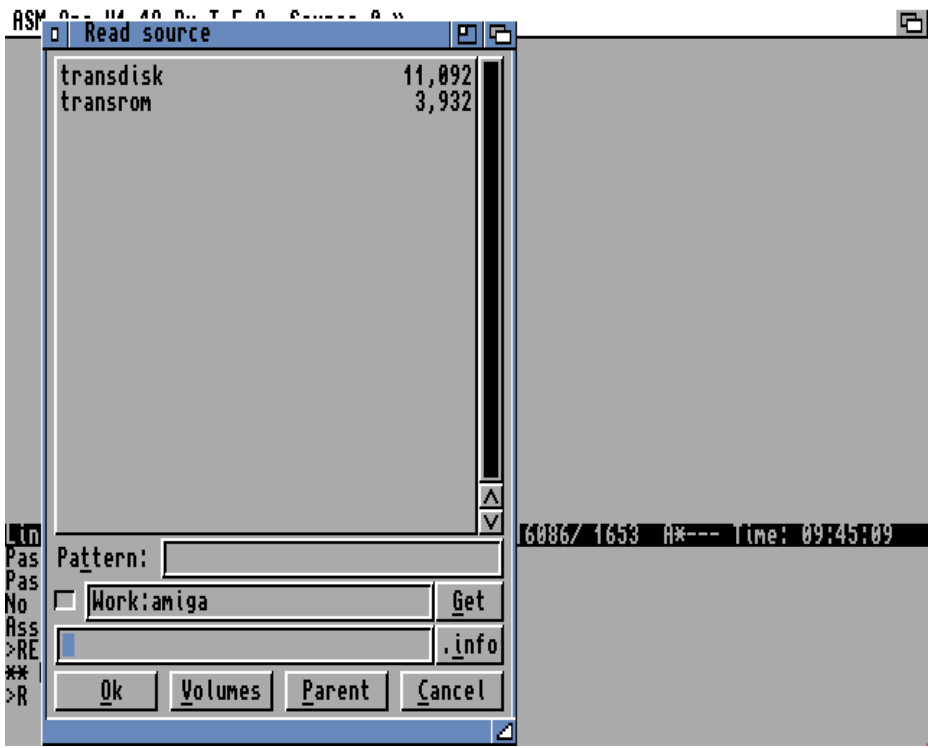
The cursor will jump below the status line and from now on you will have access to other top menu options even though the program is still visible at the top of the screen. Look out for the long menu called "Project":

Project	Assembler	Commands
Project info	=P	
Old	0	
Read	»	.48 Motorola 68k/PPC Macro Assembler (KS 2.x/3.x) ---
Write	»	
Insert	I	8k Coding by Rune Gram-Madsen (1990-1991)
Update	»	Coding and PPC Coding by T.F.A. (1991-2002)
Zap Auto-Alloc	ZA	ease, 31-12-2002 (Rev.482)
Zap File	ZF	
Zap Includes	ZI	
Zap Source	ZS	bl/Abs>Chip
		KB>200
Add WorkMem	=M	
About	#	
Quit/Restart	!	
Quick Quit	!!	

This is a typical menu found in almost every Amiga program. We have access to basic disk operations here, but this time they are separated into several subsequent functions using the sub-menu.

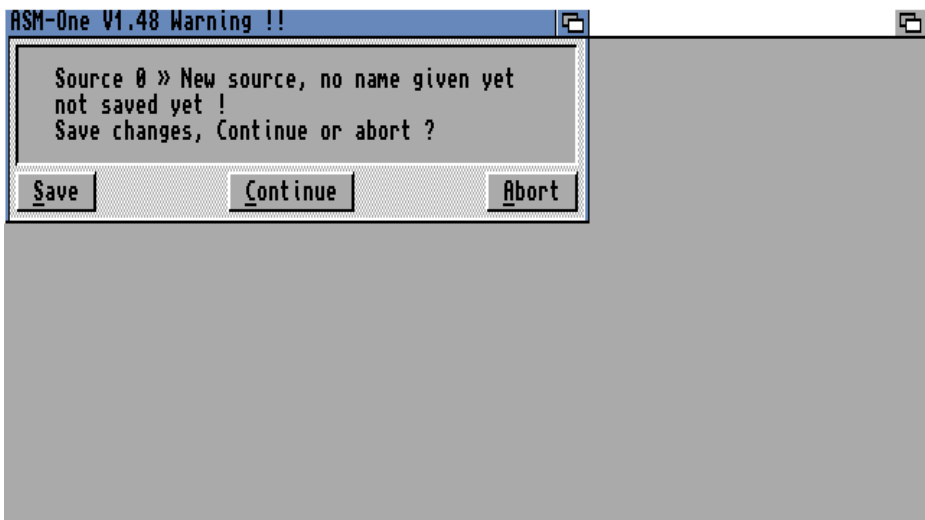
First, note the "Read" and "Write" options. Each of them contains next options, but for now the position will be interested in the "Source" position. It means loading ("Read") and saving ("Write") the program entered. If you want to continue using the command line, you can instead use two commands - R and W.

When saving, a typical selection window will appear on the screen as in the following illustration:



Select the directory in which you want to save your program, enter its name and confirm the operation with the "Ok" button. There are no difficulties here, the window works identically to other programs run on Workbench.

The situation looks a bit different when you use the option to load the program, or "Read" or the R command. An additional window may appear on the screen:



It means that the program has not been saved to disk and you can lose it. If you want to proceed without saving, select the "Continue" field. When you want to save the result of your work, use the "Save" buttons. The window is unusual because it allows you to select the save function without having to return to the top menu, as in most other typical programs.

However, this does not change the function with one caveat - after the recording has been made, "Asm-One" will immediately show another window to read another file.

At first you may be surprised by such program behavior, but over time you can get used to it. If you are not sure which function is currently called, always observe the title bar of the selection window. There you will find two information on it:

- | | |
|-----------------------|------------------------|
| - Read source | - loading the program, |
| - Write source | - saving the program. |

Also note that the menu has no equivalent of the "Save As" option, because the default save function always allows you to change the file name on the disk. So you can say that the "Save as" option is called each time.

However, if you save the file repeatedly, "Asm-One" will ask you to confirm the operation of overwriting the file, using another small window, which does not necessarily have to appear in the corner of the screen:



To save the file, select the "Overwrite" field. After the operation of reading or writing the file, a new message will appear on the command line, as follows:

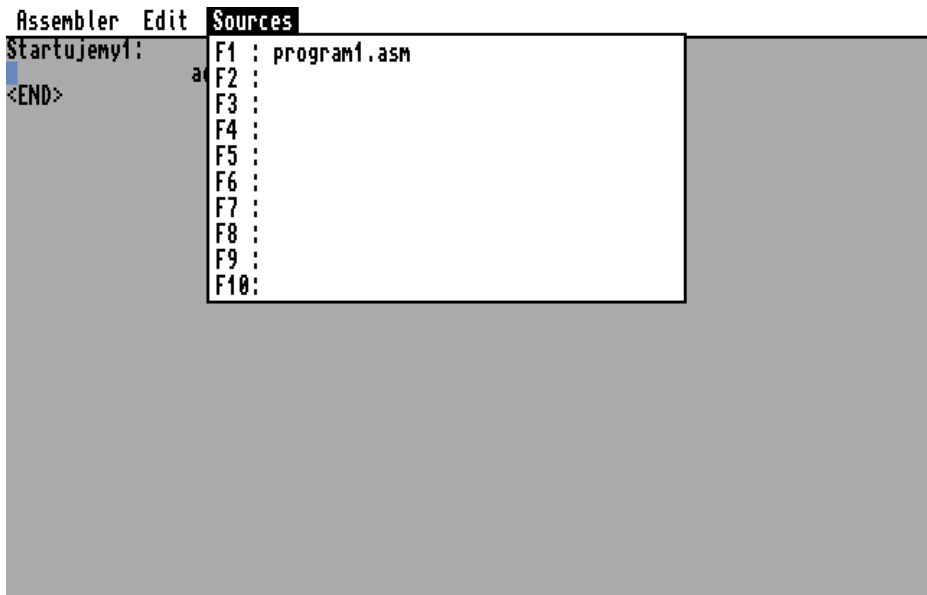
```
Line:      2 Col:   1 Bytes:    28 Free:16086/ 1653 A*--- Time: 09:45:09
Pass 1..
Pass 2..
No Errors
Assemble Time: 00:00:00
>RE
** Not done
>R
** Not done
>W
** Not done
>w
** Not done
```

```
File Length =      72 (=00000048 )
```

This means the size of the file, which is given in decimal and hexadecimal. This line has no effect on the editor, it is purely informative.

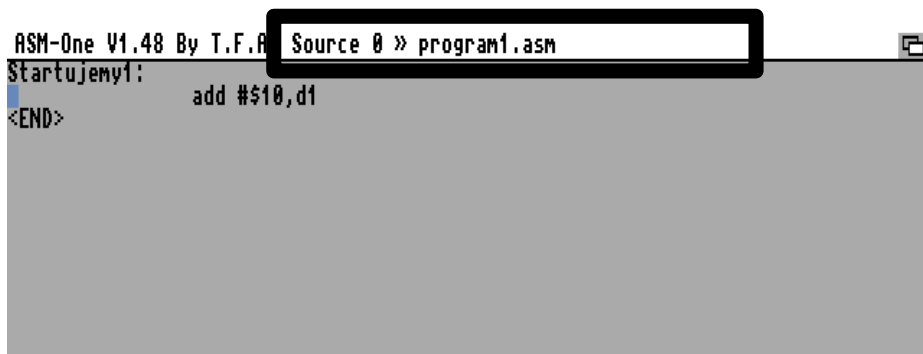
WORKING ON MULTIPLE FILES

When writing more complex programs, a good solution is to break up different parts into separate files. It may be uncomfortable in an ordinary editor, but "Asm-One" contains special functions designed to support many programs at the same time. To do this, use the top menu named "Sources", which is available on the editor screen:



At the beginning all items are empty, but after saving the program the first new item will appear on the disk. If you want to change to another program, use one of the function keys F1 ... F10. Switching occurs automatically without any additional messages. On the screen you can see the same editor screen but different content - if it is entered, of course.

Fortunately, you do not have to "manually" check which program is currently visible. This information appears on the title bar of the screen. Earlier, we did not pay attention to it, now it is worth looking at:



Next to the name "Asm-One" and the version number, an inscription similar to the following is available:

Source 0 >> program1.asm

The number next to the word "Source" indicates the number of the active program, i.e. the one currently being edited. The name after the arrows is the file under which the program is saved on the disk. Please note that you can not see the access path here, thanks to which the record is most readable, provided that you will properly name your files. It is worth remembering.

If the program is not yet saved on the disk, there is no information next to the arrows, but it will appear automatically after calling the "Write" function. Remember that the active program can only be switched in the editor window - this function will not work on the command line.

Please also note that after entering the wrong line, it will be automatically deleted when you press one of the function buttons. In other words, an incorrect record is deleted when the active program is switched to any other one. This function may be uncomfortable during learning, so at the beginning the best solution is to limit editing to one file. Otherwise, you will lose misspelled lines, which can be annoying.

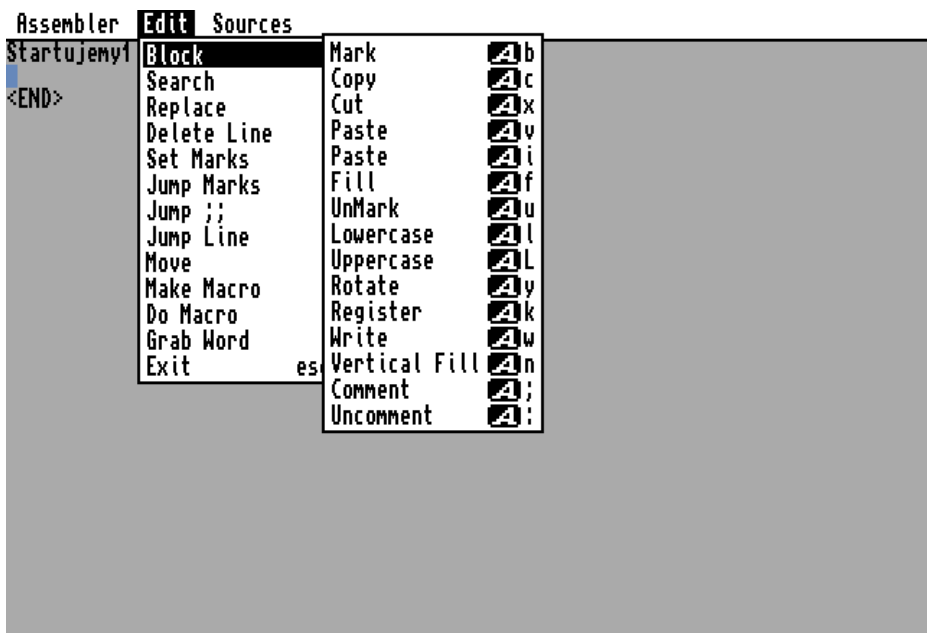
When working on many files, the symbol is also more important:

<END>

which indicates the end of the program. Theoretically, you can not enter the commands below, but when switching between active "screens", it may happen that the symbol will be inserted incorrectly. This will result in the inability to change the content of the program. To eliminate this problem you have to use functions related to operations on program fragments, i.e. blocks. We will discuss this in the next section.

OPERATIONS ON BLOCKS

As in any advanced editor, also in "Asm-One" you can use blocks to perform operations on longer parts of the program. The commands are located in the upper menu called "Edit" under the heading "Block". This is another function with a lot of options to choose from:



Remember that the menu is available to select the "Editor" function, ie on the screen where we enter the program. The most important option is "Mark", which marks the block. After using it, move the cursor to a different place, preferably with the mouse, but it is also possible with the cursor keys.

Check out the fragment of the program highlighted in the editor:

```

lea    DosName,a1
clr.l   d0
movea.l _AbsExecBase,a6
jsr     LV00openLibrary(a6)
move.l  d0,DosBase
tst.l   d0                                ; Ought to make things secure.
bne     1$                                ; Got DosBase
rts     ; Can't even get Dos library, let's get out of here!!!

```

```

clr.l   d0
movea.l _AbsExecBase,a6
jsr     LV00openLibrary(a6)
move.l  d0,IntuiBase

```

```

beq     error                                ; Ought to make things secure.

```

```

lea     GraphicsName,a1
clr.l   d0
movea.l _AbsExecBase,a6
jsr     LV00openLibrary(a6)
move.l  d0,GraphicsBase
tst.l   d0                                ; Ought to make things secure.
beq     error

```

```

movea.l IntuiBase,a6
lea     MyNewWindow,a0
jsr     LV00openWindow(a6)                ; Error checking

```

```

Line: 46 Col: 10 Bytes: 6295 Free:16084/1653 -*-B- Time: 10:10:52

```

Please note that the block should be marked by moving the cursor "forward" rather than back. This is a limit resulting from the times when "Asm-One" was created. In most newer programs, blocks can be created in a more flexible way. Once you have a block created, you can now perform a series of operations on it. Most of them are similar to those found in programs such as "Cygnus Editor", "GoldED" or even the simplest "Ed". Here's a brief overview:

- Copy, Cut, Paste, Insert

Typical features found in many programs for Amiga. They allow you to

copy, cut and insert memorized text fragments, that is, after creating a block. The last two functions are interchangeable in different versions of the editor. Most newer versions have the "Paste" option, but it works the same feature as "Insert".

- UnMark

Turns off block lighting without copying content to memory. This option should be used primarily when you select a part of the program unnecessarily. Note that after calling "Copy" the backlight is also "turned off", but then the text is temporarily stored in the computer's memory.

- Lowercase

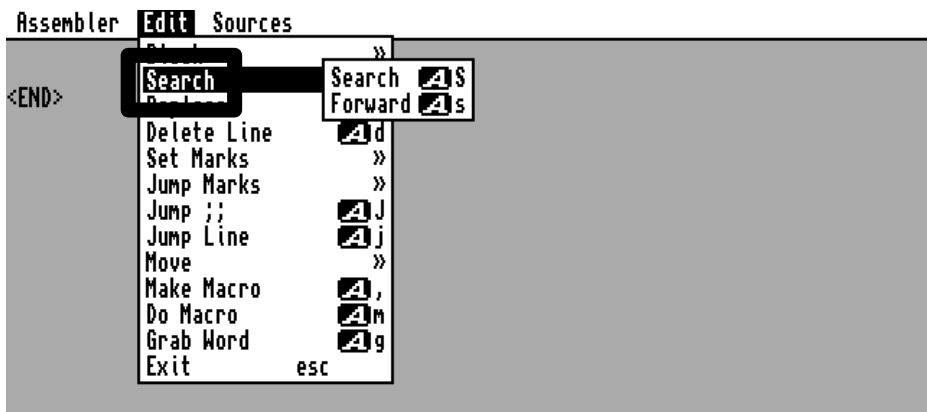
Converts the spelling of a block of text into BIG letters. In most cases, it does not matter to the functions being called.

- Uppercase

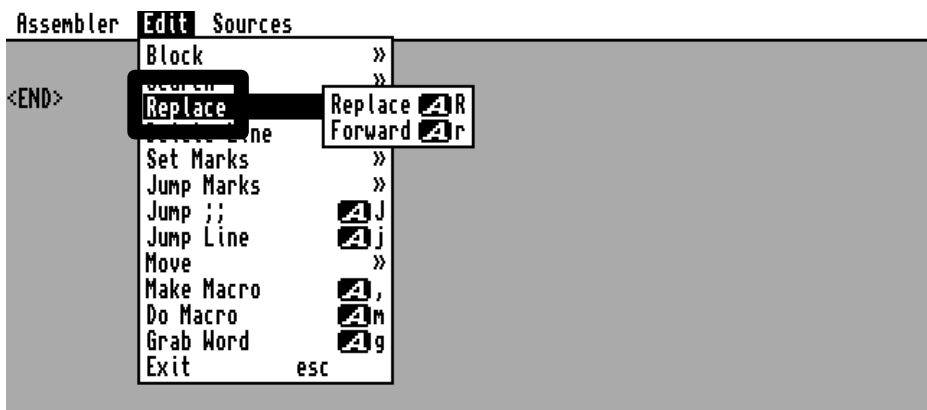
It works contrary to the previous function, namely converting the spelling of the highlighted text into lowercase letters.

SEARCHING THE PROGRAM

You can search the content of the program to find important passages, variable names, labels and other items more quickly. To make this possible we have to use the twin options visible in the same top menu "Edit", but now within the "Search" function:



or „Replace”:



Both have two very similar options. The first option activates the possibility of entering text to be searched for or exchanged. The second allows you to search for the same text string without re-entering.

Please note that text input is not in a new window, but the cursor will appear on the title bar of the "Asm-One" screen. It may look like this:



In some versions of the program, the cursor may appear only when you enter the first character of the searched string. It does not change the function, in addition, you will always see the word "Search for:" first. After pressing the ENTER key on the bar, the text will be changed and you will see the following inscription:

Searching..

If the string is found, another string will appear in the same place:

found

and the cursor will automatically be moved to the correct part of the program. However, if the entered string does not appear in the program, the following message will appear on the display:

Not Found

Please note that you can run search options many times without entering a new string. All you have to do is use the "Forward" option, which you can see in the previous illustration. This is a typical option of most editors, but in this case you can use it very conveniently using the keyboard itself. Just use the following key combinations:

RIGHT AMIGA + S

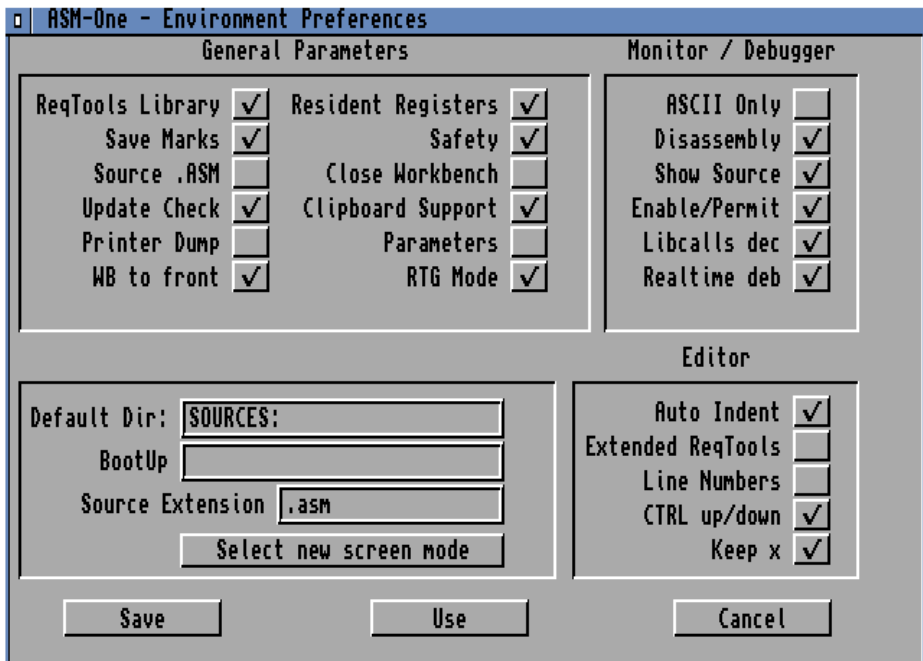
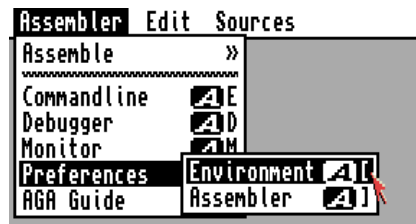
or

RIGHT AMIGA + SHIFT + S

No new window appears on the screen, and the service is always performed on the title bar of the screen. This is especially important when writing a longer program, where many complicated commands and symbols occur.

SETTINGS

In the editor window you also have a number of configuration options. They are available in two windows called using the "Environment" and "Assembler" options from the top menu with the same name "Assembler". To access both windows, first select the first function. We will not talk about absolutely all possibilities of preferences, but it is worth to be interested in the functions that affect the program and the behavior of the editor. The window available after calling the "Environment" option takes up almost the whole screen and looks like this:



The options are divided into four groups. Most of them have "marker" type boxes, i.e. square frames, which indicate "turning on" or "turning off" the function. To activate any field, just move the mouse pointer and press the left key. The service is not different from the elements visible, for example, in the Workbench preference programs.

- Display mode

The editor can work using different display modes. In older versions, you can find an option called "Interlace" with which the program starts using interlaced mode. In the basic form, this means the following mode:

PAL: Hi Res Laced

which is part of the standard Workbench configuration, so it is visible in the preference program "ScreenMode". An additional function is:

1 Bitplane

thanks to which it is possible to save the memory used by "Asm-One". After turning it on, the screen will use 1 bitplan, i.e. only 2 colors will be used.

Due to the above, the cursor will have the same color as the program's text, which is why the service is a bit difficult. On less extensive Amiga configurations, however, this is a very useful function, and sometimes even necessary, to run the program introduced or to test the operation of some functions.

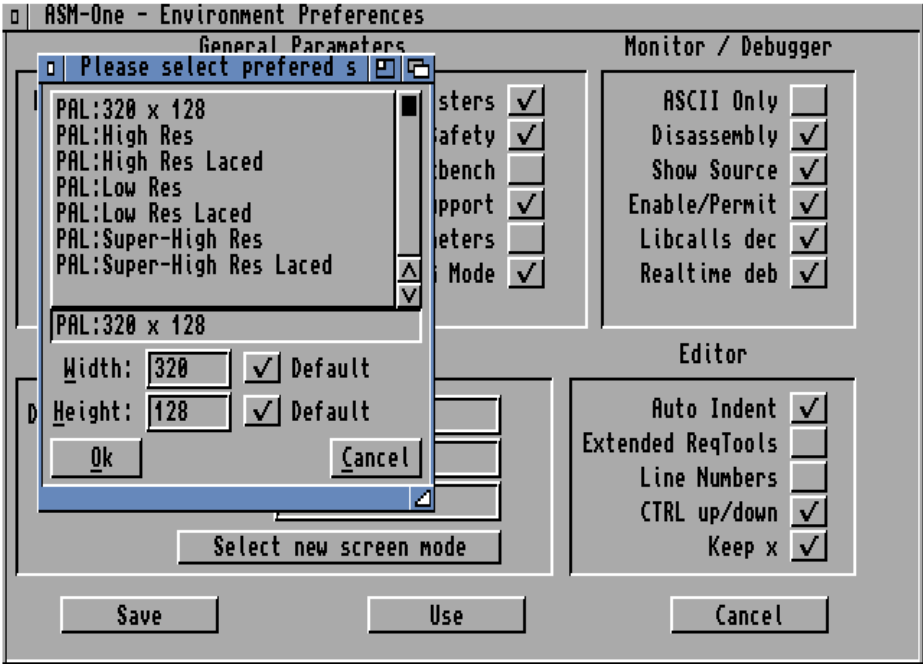
The new version of "Asm-One" does not have the above-mentioned items, instead, in the bottom part of the window, the field marked as:

Select new screen mode

After selecting it, the screen will display a typical window for selecting the display mode, where you can specify any operating parameters. Remember,

however, that the list contains items compatible with the active drivers stored in the "Devs" and "Monitors" directory on the system disk.

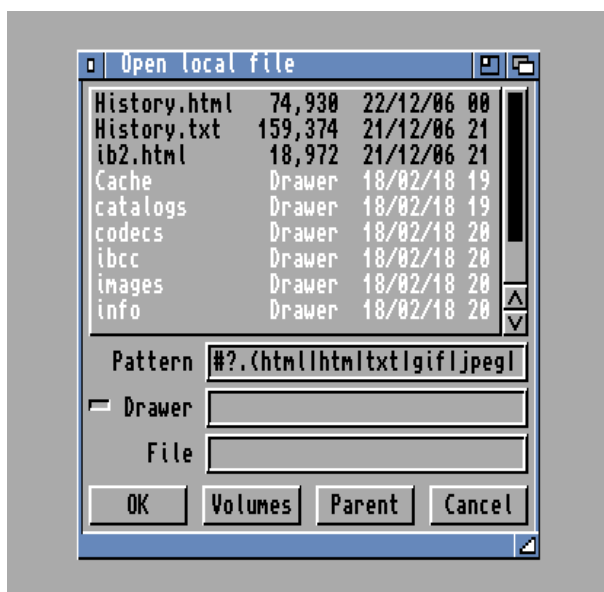
See how an example window may look like (configuration window in the background):



Select any item from the list, then use the "Ok" button at the bottom. By using the "Default" fields you can change the size of the screen, but we do not recommend it, because then you will use more memory. In addition, the additional area will not increase the screen resolution, but you will only get an extended work area that can be scrolled after moving the mouse cursor in the right direction.

- Selection windows

Please note the "ReqTools Library" option. Thanks to her, "Asm-One" will use the "ReqTools" library, which creates selection windows. We also wrote about this topic under the heading "Installation on hard drive". A typical window looks like this:



It is convenient to use, but the most important advantage is that it has low memory requirements. A single selection window takes only a dozen kilobytes.

If you did not have the "ReqTools" library installed, disk operations are of course possible, but it is difficult to do. Instead of a convenient window, only the following message will appear:

Reqtools.library not found

and the ability to manually enter the file name, preferably along with the full path - in the next command line. This situation is presented in the next picture:

A screenshot of a terminal window titled "ASM-One V1.15 By T.F.A. Source »". The window displays the following text:

```
---- ASM-One V1.15 MC680x0 Macro Assembler (KS 2.x/3.x) ----  
Original coding by Rune Gram-Madsen      (1990-1991)  
Additional coding by T.F.A.              (1991-1992)  
Additional 680x0 coding by T.F.A.        (1992-1993)  
Release date 11-03-1993 by T.F.A.  
  
ALLOCATE Fast/Chip/Publ/Abs>Chip  
Reqtools.library not found !!!  
>r  
FILENAME>  
** Not done  
>
```

A black rectangular box highlights the error message "Reqtools.library not found !!!" and the subsequent input lines ">r", "FILENAME>", and "** Not done".

This type of handling is inconvenient, but you can be forced to do it on the simplest Amiga configurations. For example, a computer may have only 512 kilobytes of memory and after loading the program, it will not be possible to call the selection window.

When using only a floppy disk drive, there may also be insufficient space to install additional system libraries. That's why you should know how "Asm-One" behaves also when you do not have an extensive Amiga configuration and a hard disk.

- Settings related to the Workbench

Another function that allows you to increase the amount of free memory is to close the Workbench. Just select the field "Close WB" or "Close Workbench" visible in the left part of the settings window. Remember that it will not be possible to disable Workbench if other programs are running on its screen or additional windows are open.

Consider whether you really want to work only on the "Asm-One" screen, because then you limit the work using the multitasking system of the Amiga.

Please also note that in older versions of "Asm-One" the preferences window will not have a "Use" button and you may not know how to call the set configuration.

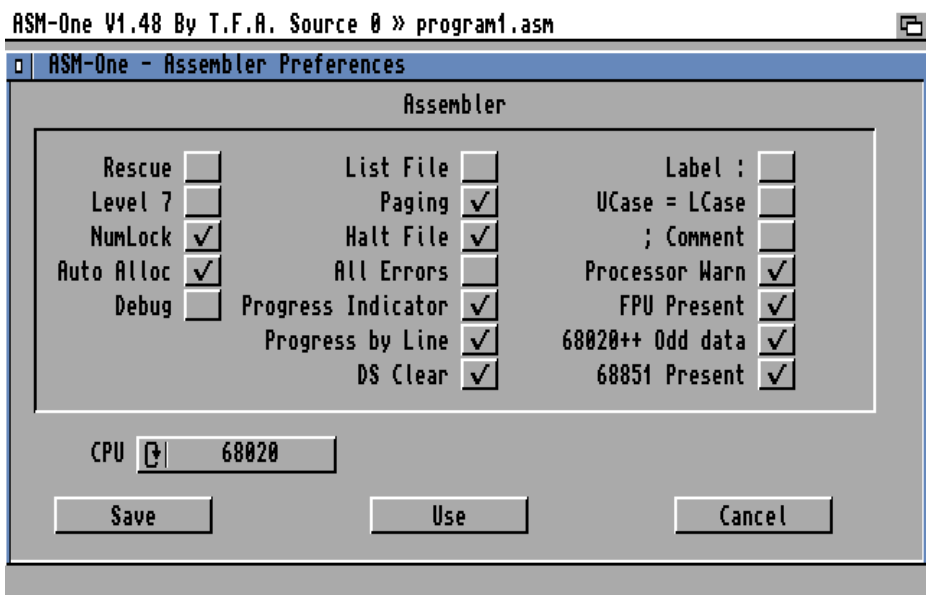
The fields visible in the bottom part of the window - "Load" and "Save" - are used to save the settings, but nothing changes on the screen. Seemingly there is no option to test the configuration. However, all you need to do is close the "Control panel" window with the usual button in the upper left corner. The configuration will be automatically called according to the status of all fields visible in the window.

This way of handling is another unusual element of the editor, because in times when the Amiga graphical interface was created, in many places it was not standardized and was subject to changes. In addition, while working in the operating system environment in version 1.3, programmers had to independently program many functions not available on the Workbench.

From here you can meet with unusual behaviors of windows or buttons. The solution is to run a newer version of "Asm-One", for example 1.48, which is available on the Aminet.net website. As discussed at the beginning of the "Editor" chapter.

ASSEMBLER SETTINGS

The previously mentioned upper menu option "Preferences" contains the second item called "Assembler". Hidden here are the configuration functions related to the details of the machine code, and therefore your program. First, look at the window:



The option called "Level 7" allows you to stop the program at any time. It can be used, for example, at infinite loops, which I write about in a separate section.

After activating the "NumLock" function, the function keys of the numeric keypad will be activated. They are described at the bottom of each key and mean:

-
- 7 - Home
 - 8 - Up
 - 9 - Page up
 - 4 - Left
 - 6 - Right
 - 1 - End
 - 2 - Down
 - 3 - Page down

In case of a large number of failures or testing of the program's operation, you can enable the "Rescue" option. Thanks to it, a potential failure that normally results in computer suspension should be intercepted so that you can continue to use the editor. Building the Amiga system and operating directly on the processor means that this function will not always be effective, but in many cases will allow you to use the basic upper menu options to, for example, save the program after changes. Thanks to this, you will not lose the effects of your work.

When you use many similar names in the program, you can distinguish them with another spelling, for example:

StartOne

and

StartONE

Such a record will help in analyzing the program. By default, both entries will be treated as two different ones, because the editor distinguishes the spelling of letters.

You can change this by enabling the option named "UCase = LCase" visible in the right part of the window. If it is active, similar entries will be treated as one. Remember that this function can significantly affect the program, so use it carefully.

When entering more complex lines, it may happen that some fragments will be treated as a comment. By default, "Asm-One" assumes that the content of the comment will be written to the right of the line, after the SPACE sign. If you enable the option marked "; Comment" you will always search for a semicolon character that should start a comment. This will eliminate the problems associated with incorrect recognition of different parts of the line.

Another option marked "AutoIndent" makes it easier to handle when using so-called "indents". We wrote about it under "Entering and starting the program". After activating this function, the SPACE or TAB signs you have entered will be automatically copied to the next line after pressing the ENTER key.

Therefore, you do not have to set up the command positions, but the editor does it "for you". This is especially important when the program is more extensive and you use more different indentations or when their size is large.

You can also make each line be automatically numbered in the editor. It does not change the program or its content, but it allows you to better understand the current position of the cursor. To make this possible, enable the option named "LineNumbers".

The configuration window, depending on the version of the editor, may contain a larger number of different functions. For now, we present the most useful, we will talk about others later. Remember that you should not change parameters when you do not know what effect they cause. In addition, knowledge of machine code should be acquired slowly in order to understand all the stages of the program's development and the principles of the processor's operation.

THE FIRST PROGRAM

- Numbers

We have already talked about different numerical systems. It's time to learn how to use them in the program. If you want to use values saved in decimal places, put a "#" in front of the number, ie, for example:

#15

When you need to use a hexadecimal number, add the dollar sign:

#\$15

Of course, in this way we will write completely different values, because the number 15 in hexadecimal format is 21 decimal.

You can also save numbers by entering both previous symbols, for example:

#\$00000020

#\$B5

#\$11

- Variables

In each program we need to use data - text or numbers. We can define them as so-called "variables" and perform many different operations on them. This can be compared to the data we use when solving math problems, substituting specific values for the formulas. Likewise, it will work in your program, but the way you use variables is completely different than in high-level languages, for example in Basic.

We must use this already mentioned processor registers for this purpose. It is in them that we store numbers, as well as perform operations necessary for the work of the program. Therefore, it can be said that the registers play the role of variables using the D0, D1 or D2 type names. After saving the data, the next step is to read the result and use it in accordance with the assumptions of our program.

- Comments

In the program, you can post comments to facilitate the analysis of the entered lines or separating separate parts. They are optional, so you do not have to use them. They can be compared to the REM instructions in Basic.

However, Assembler differs in the way of marking comments. In the "Asm-One" editor, lines that start with an asterisk are treated as a comment in the entire line, you can also put multiple "*" or comment characters at the end of the line using the semicolon character. See how it should look like:

```
ASM-One V1.15 By T.F.A. Source » Screen.a
clr.l    d1                ; Return code for CLI
movea.l  DosBase,a6        ;
jsr      _LV0Exit(a6)      ; Clean exit from WB also

* Data section
***** Screen data

MyScreen:
dc.l 0

MyNewScreen: dc.w 0,0,320,256 ;size filled in later
              dc.w 4          ;depth
              dc.b 0,1
              dc.w 0          ;viewmodes
              dc.w CUSTOMSCREEN
              dc.l 0
              dc.l ScreenTitle
              dc.l 0,0

***** Window data

MyWindow:
dc.l 0

MyFlags    EQU SMART_REFRESH!ACTIVATE!WINDOWDEPTH!WINDOWDRAG!WINDOWCLOSE
MyNewWindow: dc.w 0,30,250,100
              dc.b 2,5
              dc.l CLOSEWINDOW ; IDCMP Flags

Line: 69 Col: 1 Bytes: 2756 Free:13446/1751 ----- Time: 01:05:15
```

In principle, any text "on the right" separated by a space character can be treated as a comment, but the best solution is to use semicolons. Thanks to this you will never make a mistake while analyzing the program, even if you entered it long enough that you do not remember the exact content.

Using comments can sometimes cause trouble, so please note the following option:

; Comment

located in the configuration window. We write about it in the section titled "Assembler settings".

- Labels

Labels do not have to be used, but if you want to use them, they should be placed at the beginning of the line or in separate lines, before specific parts of the program. The name must be ended with a colon to allow "Asm-One" to recognize the label correctly. This allows you to mark different parts, as well as perform "jumps", which you will read, among other things, under the heading "Loops".

In practice it can look like this:

```
File1:      add #$10,a0
```

or

```
File1:  
      add #$10,a0
```

Of course, the line after the name of the label is an example and will vary depending on the purpose of your program. Please note that the name should be saved in the columns preceding the names of the instructions. For example, if the program looks like this:

```
File1:  
add #$10,a0
```

you call an error. In the case of Asm-One, the visual form of the program is as important as the commands, variables and other symbols necessary for the operation of the machine code.

Labels can also appear as arguments to the instructions, as we'll show in many examples later in the book. Usually they are used to determine the location of the "jump", for example to a subroutine.

Labels can also be used to specify the space intended for variables or other data needed to run the program, as well as parts important for the testing of algorithms. For this reason, you should use simple names that will not mislead you even if you check a program written much earlier. Of course, you can use almost any markings, but it's better to keep the naming rules clear. Such a program is much easier to analyze, especially when an error is difficult to locate and correct.

- Stack pointer

You will certainly encounter situations in which you will need to store important information used in the program. The "stack" is intended for this purpose. This is a memory area where you can temporarily store data. You can decide in which part of the processor's memory the stack will be located, and change its address at any time.

At this point, we need to introduce another concept, the so-called stack pointer labeled with the abbreviation SP. Its role is played by the A7 address register.

You have to decide where to put a stack in your memory. To do this, use a line similar to the following:

```
movea.l    #$00100000, a7
```

It sets the address of the stack to 100000 using the A7 register. The stack pointer is also related to the loops I write about at the section under the same title.

- Instructions and mnemonics

680x0 processors have a specific set of instructions. They have different functions, but are "understood" by the processor in a strictly defined way, namely - in a binary form. Here is an example:

0011 0000 0011 1100 0000 0100 1111 0000

Of course, for us it is completely incomprehensible, fortunately we do not have to learn this way of working. In return, we will use the so-called mnemonics. They are just a few-letter words that mean a specific action to be performed by the processor, for example:

ADD

or

SUB

For the purposes of the book, we will use the name "instruction" interchangeably. The detailed use of the various instructions will be discussed later. Regardless of them, it is very important that you understand the basic concepts.

It is worth noting that when starting the program it is converted into a form directly understandable for the processor, i.e. as in the above example. This is what our Assembler and the editor in which we write the program deal with.

- Operators

When writing a program, we will use many symbols and characters that are used to create so-called expressions. These may be, for example, mathematical calculations or more complex functions. You can use the following basic operators:

- mathematical:	+	or ADD	- addition
	-	or SUB	- subtraction
	*		- multiplication
	/		- division
- logical:	&	or AND	
	OR		
	EOR		
- comparison:	>		- greater value
	> =		- value greater than or equal to
	=		- equal value
	<=		- value less or equal to
	<		- smaller value
	<>		- value lower or bigger

In the further part of the book we will deal with the practical use of these symbols.

- Wykonywanie programu

Before we write any program, we need to know how it will work in general terms. Instructions are written one below the other and the program is run the same way. Take a ready example:

```
Suicide:                ; If ram is running low, we'll give ours up.
    lea    0,a0          ; To beep every screen. This is not a very
    movea.l IntuiBase,a6  ; polite thing to do, but low ram is a disaster
    jsr    _LV0DisplayBeep(a6)
    bra    exit          ; Now leave

Convert:
* Inputs: address in a0, data i d0.
    ; If data is starting in ASCII before the address, data spent.
    move.l  d0,d1
    andi.l  #5FFF0000,d1 ; Divide long word by 10 may cause overflow.
    swap    d1
```

In this situation, the MOVE.L statement will be executed first, followed by ANDI.L, and finally SWAP. This may seem obvious to you, but you should also know that there are instructions that can change the order in which the lines are executed, even though they will still be written one under the other. Therefore, not every part of the program can be run on a "line by line" basis.

The meaning of words visible in the illustration will be explained in the following chapters of the book. However, you can read about the instructions that change the order in which the lines are executed in the "Loops" section.

SIMPLE OPERATIONS

- Adding values

Programming is best started with simple operations, which we will then gradually expand. Let's start with math operations, which in high-level languages (like Basic or Pascal) do not require a wider commentary. However Assembler has it to himself that it requires more attention.

Let's start with the operation when we want to add a few numbers to each other. The basic instruction is in this case the word ADD, by means of which we can add a value to a specific register. This may look like this:

add.b #5, d0

This line will add the value 5 to the register with the symbol D0. Please note that we use the ending ".b", which means "byte". Alternatively, the ".w" and ".l" terminals can be used. What are the differences? We explain in the point titled "Numbers".

Repeat this operation, which will add another value to the same registry. However, the ADD instruction does not create a "new" value, so the previous contents of the registry will not be deleted.

In many cases it will be useful, but from time to time you will definitely want to reset the value stored in the registry. How to do it, read the section entitled "Cleaning the contents of registers".

Of course, in the same way you can put a value in another register, for example - a data register with the symbol D1. To make this happen use the following line:

```
add.b #1, d1
```

When we have two "variables" we can add them to each other by entering two register names, for example:

```
add.b d0, d1
```

We must remember that the ADD statement changes the contents of the target register, and the source value will remain unchanged. Therefore, in this case, the D0 register will keep the number 5, while the D1 register will store the totaling result, ie the value 6.

The numbers stored in registers can also be saved in a specific place in the computer's memory. Just use a record similar to the following:

```
add d0, $00001012
```

Thanks to this, the contents of the D0 register will be "copied" into memory, exactly to the address \$ 00001012. However, it is not possible to add the contents of two memory addresses directly. When you apply such a record:

```
add.b $00000010, $00000015
```

you call an error. But you can copy the data using a different MOVE instruction. How to do this you can read in the section titled "Copying data".

Please note that depending on the length of our "variable", information will be stored in subsequent memory addresses. The data will be differently saved after applying the ADD.B instruction, and differently after entering the line containing the word ADD.W.

We discussed the differences in action earlier. These rules apply when using all instructions. We emphasize this many times, because without understanding the principles of information storage, you will not be able to write effective programs.

You can also use the word **ADDI** (and. Immediate), which works similarly, but adds the so-called "immediate" variable, which is directly the number given in the line with the instructions. The method of use is the same, for example:

```
addi .b #48,d0
```

The last possibility is the so-called "quick" addition using the **ADDQ** (Quick) command. You are limited here, because you can only enter values from 1 to 8. This command uses an immediate variable, like **ADDI**, and can be useful when optimizing your program, which we will talk about later. In a ready-made program, the line with the **ADDQ** instruction may look similar to the previous one.

- Subtraction of values

Decreasing the value is possible with the word SUB. Works similar to the previous one, we can use the following line, for example:

```
sub.w #32, d0
```

that the value 32 is subtracted from the contents of register D0. This instruction is governed by the same rules as ADD, whereas it has the opposite effect. We can also operate on the registers themselves, that is, subtract the numbers that we store in them. Let's look at another example:

```
sub.l d0, d1
```

This will subtract the "long word" in the register D0 from the contents of register D1. There is also no problem with deducting data using a specific memory address - on the same principle as before:

```
sub.w d0, $0000102E
```

We still can not perform operations directly on two memory addresses, therefore instead of using an incorrect entry:

```
sub.b $00000120, $00000124
```

it should be replaced with two lines using data copying, i.e. with the MOVE instruction. We will deal with this type of operation at the point entitled "Copying data".

As with the addition of values, it is also possible to use the words when subtracting:

SUBI - "immediate" subtraction

and

SUBQ - „quick” subtraction

The way in which you should save them in the program is no different. Their operation is analogous to the ADDI and ADDQ instructions, which are discussed in the section "Adding values".

- Multiplication of values

Calculations other than addition and subtraction are easy. You must use other instructions, but the general usage pattern is similar. Multiplying the value stored in the register causes the Mulu command, for example after the line has been executed:

```
mulu.w    #$0010,d0
```

the contents of the register D0 will be read, followed by the multiplication by the number 0010. The result is stored in the same data register.

Note that the target (decimal) argument must be a data register, while the source value may not only be a direct number, but also a different data register or memory address. For example:

```
mulu.w    d1,d0  
mulu.w    $00000010,d0
```

It is also possible to use the address register in such a way that its contents point to a specific address in memory. You must then write its symbol in brackets, but this does not change the general principle of the function. For example, like this:

```
mulu.w    (a0),d0
```

However, you can not use the address register directly in this way, i.e. as below:

```
mulu.w a0,d0
```

In addition, in this case you must only use "words", you can not use bytes or long words.

The second way is to enter a MULS instruction that behaves similar to MULU. Here's another example:

```
mul.s.w    #$0010,d0
```

As before, the value from the D0 register is multiplied by the word 0010. However, the number in the register will be treated as negative, therefore you will receive a different calculation result - also with a negative sign. This is the only difference between the two commands.

- Dividing values

To perform a splitting operation you can use two instructions - MULU and MULS - just like in the previous section. The first one divides the long word by the source argument and we use it in the same way as for multiplication, for example:

```
divu.w    #$0002,d0
```

The quotient is stored in the data register - in this case the symbol D0. The remainder of the division will be placed as the top word of the same register.

If it turns out that the result of the operation is greater than the word, it can not be saved, consequently the value of the register will remain unchanged. You must use the data register, while the source value may be a direct number, a different data register or a memory address. The same rules apply to multiplication operations, so they are easy to remember.

Likewise, we also have the option of entering a DIVS instruction with which you can also share negative numbers. Note, however, that the sign value in the register determines the result of the operation. In other words, if we enter the following line:

```
divs.w    #$0002,d0
```

The quotient will be a positive value only if the contents of the D0 register also have a "plus" sign.

For this reason, you may have to perform calculations in a specific order in your program so that you do not get a number with an opposite sign than you expect.

We mentioned earlier that the quotient can not be larger than the word, so it means that its maximum value is:

FFFF

If you get a higher result of the operation, the statement will not put the answer in the registry. Its contents will also not be deleted or incorrect data will appear. The value in the registry will remain unchanged, and the program will continue to work without the division.

Another situation is the attempt to divide by zero, which is obviously impossible to do. However, if such an operation occurs, for example after using a line similar to:

divu.w \$0000,d0

the processor will run the "exception" mechanism, ie a special procedure designed to handle errors and other "exceptional" events. It is a more complicated topic, which is why we will discuss it later.

The maths rules are clearly defined, but if you want to make writing a program easier, you can assume that the result of such a division is always zero. All you have to do is remember that this can change the calculated calculations, that's why they all depend on their order and purpose.

- Single bit operations

So far, we've used instructions that change data in the form of bytes, words or long words. There are also possible more precise modifications, within individual bits, and thus in much smaller units. Three basic instructions are used for this purpose: BSET, BCLR and BCHG. They all have a very similar way of use.

The first BSET command sets the specified bit number to 1. You can use them like this:

```
bset .1    #$0E,d0
```

The above line will cause the bit with the number 0E (14 in the decimal system) to be set to 1. To better understand the long word we can present in the following form - before setting the bit:

```
00000000 00000000 00000000 00000000
```

and after executing the line with the BSET instructions:

```
00000000 00000000 01000000 00000000
```

Of course, the maximum bit position in the "long word" is 1F (31 decimal), while the source value can be higher. In this situation, the command will work a bit differently.

For example, this entry:

```
bset .l    #$27, d0
```

means that the bits after position 1F will be zeroed and counted from scratch. Consequently, the value 1 will get bit 07, i.e. our long word will look like this:

```
00000000 00000000 00000000 10000000
```

This item results from the calculation of:

```
27 - 1F
```

Keep in mind that the first position is 00, so our "eighth" bit is position 07.

If you want to use a memory address instead of a register, you can not use a word or a long word. You must use the size of the byte, i.e. the command with the ending ".b":

```
bset .b    #$00, $00FF8010
```

This limitation does not apply when using source values stored in other data registers, for example:

```
bset .l    d1, d0
```

Please note that it is not possible to use the address register in the same way:

```
bset .1    $00, a0
```

The other two BCLR and BCHG instructions should be used identically. BCLR "clears" the specified bit, that is, setting its value to zero.

The example line may look like:

```
bclr .1    #$0E, d0
```

The last instruction is BCHG and - as you can guess from the short version version - its function is to change the bit value. If 1 is written, the bit will get 0 and vice versa.

- Storing the address in the registry

A very important function is also the ability to place data about specific addresses within address registers. The LEA instruction is used for this purpose, which should be used like this:

```
lea    $20000,A0
```

As a result, the entered address will be placed in the register with the A0 symbol. To later retrieve the address information from another part of the program, you must enter the name of the registry in parentheses. Of course, this can be an argument to another command, for example:

```
move.b d0, (a0)
```

Take into account that such a record is not applicable to all possible instructions.

LOGICAL FUNCTIONS

In addition to mathematical operations, we can perform logical operations. They are quite easy to understand in high-level languages, in the case of Assembler we have to deal with them a bit longer.

The basic logic functions are:

- AND
- NOT
- OR
- XOR (EOR)

The result of the logic functions is TRUE (true) or FALSE (false). In the first case it will mean saving the value of 1 and the second - zero. To invoke logical functions, we use the instructions given above.

- AND

Let's start with the AND function. This instruction will perform an AND operation between the source value and the target processor register, as before. The result of its operation will be as follows:

<u>Source value</u>	<u>Value in the register</u>	<u>Result</u>
0 (False)	0 (False)	0 (False)
0 (False)	1 (True)	0 (False)
1 (True)	0 (False)	0 (False)
1 (True)	1 (True)	1 (True)

In other words, for the result to be the number 1, both components must also have the same value 1. In the program, the use of a function can be written, for example, like this:

```
and.l    $00004000,d7
```

The result in this case will be saved in data register D0. Remember that the numbers will be internally saved in binary (binary) form, where there are only two possibilities - 0 or 1. The individual digits will be compared and the result of the whole operation will depend on it.

This may seem complicated, so see the example of the AND function on two numbers:

```
226  
and  
103
```

In the binary system, you should write them like this:

226 = 11100010

103 = 01100111

This function is often briefly saved as the "&" character, therefore we will use the same record below. A comparison of individual digits (bits) will give the result:

1	1	1	0	0	0	1	0
&	&	&	&	&	&	&	&
0	1	1	0	0	1	1	1
=	=	=	=	=	=	=	=
0	1	1	0	0	0	1	0

As you can see, we get the number:

01100010 = 98

The value after the equal sign means again the value written in decimal. Calculating the operation of the AND logical function is not complicated but tedious. That's why you will not have to do it manually, but you should know what the principle of operation is.

In the "Adding values" section we are talking about so-called immediate variables. You can use them also using the AND function by entering the ANDI command.

Here's an example in a longer program:

```

ASM-One V1.15 By T.F.A. Source » MemTool.a
Convert:
* Inputs: address in a0, data i d0.
* Output: Numeric string in ASCII before the address, data spent.
  move.l    d0,d1      ; Divide long word by 10 may cause overflow.
  andi.l    #$FFFF0000,d1
  swap      d1

nextdigit:
  divu      #10,d1
  move.w    d1,d2      ; The very high part of the number
  andi.l    #$FFFF0000,d1
  andi.l    #$0000FFFF,d0
  add.l     d1,d0
  swap      d0
  addi.b    #48,d0      ; low byte now contains ASCII digit
  move.b    d0,-(a0)    ; Move to string
  swap      d0          ; Move remaining number back
  andi.l    #$0000FFFF,d0 ; Wipe out the digit.
  move.l    d2,d1      ; Put the high part where we need it
  add.l     d0,d2      ; This to test for the end: d2 is scratch, containing
  bne       nextdigit  ; data that have been copied to d1. We can then use it
                        ; to determine if we're done.

  rts                          ; All digits gone to the string.

* Data section
Line: 163 Col: 1 Bytes: 6295 Free:13446/1751 ----- Time: 01:07:59

```

The lines marked with the frame cause the execution of several AND logical operations, the first two using immediate variables within data registers D1 and D0, and the next one calls the function in relation to the values previously stored in the mentioned registers. These are not complicated operations, although understanding their result requires studying the principles we discussed at the beginning of the section.

- NOT

The next logic function is NOT. It works a completely different way, namely, it reverses the value of all bits. Zero will become one and vice versa, so understanding it is much easier. All you need to do is enter the processor's register:

```
not.b    d0
```

so that the content is "reversed". This command can also be used in relation to the address in memory, for example:

```
not.w    $000048C0
```

It is also possible to enter the address register, but not directly, only after saving it in the address register. We also write about it under the heading "Saving an address in the registry". Therefore, the name of the register must appear in brackets, for example:

```
not.w    (a0)
```

- OR

This function is another one that we have to think about longer. The general principle of operation does not change in relation to the AND function, but now the comparison of the two bits will result in a different result of the operation. Look at the table in the same way:

<u>Wartość źródłowa</u>	<u>Wartość w rejestrze</u>	<u>Wynik</u>
0 (False)	0 (False)	0 (False)
0 (False)	1 (True)	1 (False)
1 (True)	0 (False)	1 (False)
1 (True)	1 (True)	1 (True)

The OR command can be applied to both data registers and memory addresses. However, you can not perform operations directly between two addresses and address registers.

Here are some examples of how to correctly write an OR statement:

```
or.w    d1,d0
or.w    d0,$00003000
or.w    d0,(a2)
```

If you want to learn how this function works, you can explain it as follows. If at least one of the compared bits has the value 1, as a result of the function we will also get the same number.

While performing an OR operation, it is also possible to use immediate variables, about which we wrote earlier.

- EOR (XOR)

The last function is a disjunction marked with the abbreviation XOR or EOR. The command in Assembler should be used in the second version, ie, for example:

```
eor.w d0,d1
```

Once again the result of the operation depends on the comparison of individual bits, but now it will look like this:

<u>Wartość źródłowa</u>	<u>Wartość w rejestrze</u>	<u>Wynik</u>
0 (False)	0 (False)	0 (False)
0 (False)	1 (True)	1 (False)
1 (True)	0 (False)	1 (False)
1 (True)	1 (True)	0 (True)

The simplest way is to say that if one of the components has the value 1 and the second 0 - the result of the operation will take the number 1. Otherwise, we will get zero.

Another difference is the fact that the EOR function can not have an argument in t

```
eor.b (a0),d0
```

meaning that the address register with the A0 symbol has previously been saved and we can now use this information.

You can read more about this topic in the section "Saving an address in the registry". It is possible, however, to use the so-called immediate variable, about which we also wrote in the section "Adding values". Just enter the command named EORI, the operation of which is analogous to the "ordinary" EOR function.

- Copying data

One of the most useful instructions is MOVE, which copies data, for example to a specific processor register. Just use this line:

```
move.b $23,d0
```

that the number 23 written in hexadecimal is copied to the register D0. Remember that, unlike the previous instructions, you do not invoke the add operation in this way, only the value transfer.

Nothing prevents you from copying data between different registers, for example:

```
move.b d1,d0
```

or to a specific memory cell:

```
move.b #50,$0000103A
```

The matter is somewhat more complicated if we want to use data saved as "word" or "long word". If the memory address is odd, the information transfer operation will fail. Therefore, the following line:

```
move.w $00000067,d0
```

will cause our program to crash.

However, access to our data stored in odd-numbered addresses can be obtained by using "byte", i.e.:

```
move.b $00000067,d0
```

- Increment and decrement

These two words sound rather strange, and they simply mean increasing and decreasing the value by a unit. Such operations are possible using registers. This may look like this:

```
move.b    #$B5, (a0)+  
move.b    #$11, (a0)+
```

Please note that at the end of the line there is a "+" sign, which means an automatic increase in value. Our short program will transfer the number B5 to the register A0, and then the value written in it will be increased by a unit. The next line carries the number 11 to the register A0, and the symbol "plus" causes the increment, i.e. adding the number 1.

If we use data like "word" or "long word", the value increase will be slightly different. In the first case, the number 2 will be added, and in the second - 4. This, of course, is related to the length of the information to be recorded, as described in the section entitled "Units of volume".

So if we enter the following line:

```
move.w    $A90E, (a0)+
```

Our "word" will be changed in such a way that the value will be added 2, not 1 - as in the case of "byte".

Subtracting values works similarly, but we must use the "-" sign written to the left of the register symbol, i.e. for example:

```
move.b 2E, -(a0)
```

The value will change in the same way, so in this case the number 1 will be subtracted, because we are dealing again with the value of the "byte" type.

- Clearing the contents of registers

When the operations of changing the contents of registers will be performed in turn, there will certainly be a situation in which the removal of the number will be useful, ie setting the register to the value "zero". This can be done with the next CLR instruction, and its use is much easier.

Here is an example:

```
clr.b d0
```

It should be remembered that zeroing will apply to the value of a specific type, in this case the "byte". It means that if the value 01234567 is saved in the register, after the cleaning we get the result:

```
01234500
```

Of course it is also possible to use "words" and "long words", it is enough to change the end of the instruction, for example:

```
clr.w d0
```

The same instruction can be used in relation to memory addresses:

```
clr.b $00201FFF
```

However, it is not possible to use the "address" registers in the same way, for example:

```
clr.l a0
```

Of course, there is another way to delete values stored in such registers. All we need to do is to copy the data after "reset" in the "data" register:

```
clr.l d0  
move.l d0,a0
```

It is not complicated, but it extends the program and causes that we must use an additional "variable".

- Loops and jumps

As in other programming languages, you can also use instructions in Assembler that change the order in which instructions are executed. To use them, we must introduce another concept of the so-called "Counter" (Program Counter) marked with the abbreviation PC.

This is a special processor register for storing the current location where the processor is currently running. In this register, the address of the currently executed or next instruction is stored. We also write about it in the point of "orders". It is through modification of this register that we can use such elements as jumps, loops and subprograms.

To use this information effectively you need to learn the next JMP command. It allows you to change the meter's status, thanks to which the processor will continue working at the indicated destination. In the program, this may look like this:

```
add.w    d1, d1
add.w    d1, d0
jmp      Skok1
add.w    d2, d3

Jump1 :
move.w   d0, d2
```

Of course, the above lines are examples, but show a diagram of the JMP instructions. Remember that this is a simple jump, so the program will go to the line located after the label "Jump1" and it will continue to work as if you did not enter the line between the JMP and the name of the label.

This is very important, because in this situation some program lines may never be made. You can achieve a very similar result using the BRA instructions. It also causes a "jump" to a specific place and its use is analogous. See how it can look in the editor:

```

error:      ; OpenWindow or some OpenLibrary failed
movea.l    AbsExecBase,a6
movea.l    IntuiBase,a1
jsr        _LVOCloseLibrary(a6) ; This is not really needed
clr.l      d1                  ; Return Code always zero
jsr        _LV0Exit(a6)        ; Done
clr.l      d0
rts        ; Well, almost done. It seems Exit sends us
           ; back here. We'll return completely now.

Suicide:    ; If ram is running low, we'll give our»
            ; To beep every screen, This is not a v»
            ; polite thing to do, but low ram is a »
            ; Now leave
leal        0,a0
movea.l     IntuiBase,a6
jsr         _LV0DisplayBeep(a6)
bra         exit

```

The main difference between these two commands is memory and speed. The BRA instruction is faster than the JMP instruction, it also has a smaller size. Thanks to this, you can speed up the program as well as cause that it will require less memory. It has a special meaning - as usual - when writing programs operating on the classic Amiga 500 with 1 MB of memory.

The next instruction related to creating the loop is JSR. Its use is very similar and may look like this:

```

add.w      d1,d1
add.w      d1,d0
jsr        Skok2
add.w      d2,d3
asr.w      #$04,d0

```

Jump2 :

move.w d0, d2

However, the way of operation is slightly different. In addition to the jump execution, the return address will be remembered so that the processor can then continue using this address. It is moved by the so-called stack, which we write about in the item under the heading "Stack pointer".

However, the "return" jump is not performed automatically. To make this possible, use the additional RTS instructions. Thanks to it, the appropriate address is loaded from the stack and placed in the command counter. The processor can thus continue to work from the stored address.

Look at the scheme of using this function:

```

                move.w    d0, d1
                jsr        Skok3
                move.w    #$0020, d1
                jsr        Skok3
                jmp        Dalej
Skok3:
                add.w     d1, d2
                rts
Dalej:
                move.w    d1, d2
```

Now, after jumping to the line marked "Jump3", the instructions between the label name and the line with the RTS instruction will be executed. Then the program will return to the previous location and a line will be called causing the jump to be marked as "Next".

In other words, the following structure:

```
        jsr      Label1
        ...
Label1:
        ...
        rts
```

you can treat as a subroutine that can be executed multiple times. Such fragments are useful, for example, when you need to run multiple times the same operation or you want to divide a program into many separate parts to make it easier to analyze the operation.

As with the JMP and BRA command pairs, you can also use a faster and less memory-intensive instruction called BSR for "subroutines". The method of use is the same, but in this case we can use two versions:

```
bsr.s      or      bsr.w
```

For "words" and so-called "long words". If you have doubts about the difference, go back to the chapter "Fundamentals", where we discuss both concepts.

CONDITIONAL STATEMENTS

- CMP instruction

In the program, we can use instructions that will change the execution of functions in different situations, for example when a specific bit changes the value. It is necessary for the code to perform its functions and be able to react to the user's behavior. The function will change operation under a given condition, hence the name of this instruction group.

The basic command to be learned is CMP. It compares the source value with the contents of the data register. Take a look at the practical example:

```
cmp.b    #$20,d0
```

The above line will check whether the D0 register contains the number 20. If this is the case, the result will be a zero value. However, the CMP command alone will not cause the program to react automatically to the result. You must add another line containing the command that reacts appropriately to the result of the comparison. We have a few basic options here.

- BEQ instruction

The first is the instruction with the BEQ symbol. You can use it like this:

```
cmp.b    #$20,d0
beq.s    WeHaveZero
...
```

WeHaveZero:

...

Thanks to the BEQ instruction, a "jump" will be made - only if the result of the comparison with the CMP word will be positive (the values will match), so the "zero" flag will be included in the result.

- BNE instruction

The reverse operation to the previous one can be obtained using the BNE command. It also makes a comparison, but it does a "jump" if the compared values do not match, so the zero mark will be "cleared". If the tag has a value of 1, the line with the word BNE will be ignored. The method of use is identical.

- TST instruction

Another possibility to obtain a condition is to use the TST command, which compares the entered number with the zero value. Here is an example of use:

```
tst.w d0
```

In this case, "zeroed" will be with excess and transfer markers, which you can read about in the next section.

Please note that thanks to the previous instructions you can get the same effect. For example, by entering the following line:

```
cmpi.w    #$0000,d0
```

However, TST takes up less memory, and works faster. Therefore, if you are going to check zero values, we recommend using the TST command.

- BPL instruction

The jump can also be made when the sign marker is "off". You just have to use the BPL command - in the same way as before. Thus, this means the action in the case of a "positive" result of the comparison of values.

- BMI instruction

The last instruction with the BMI symbol is the inverse of the BPL, that is, it will make a "jump" if the result of the comparison is negative, and so if the character mark gets the value 1.

Of course, in the finished program instead of the ellipsis will be placed other instructions, above we give only the scheme of application. The general scheme of conduct is the same, you need to apply a similar structure in the program, whereas its operation depends on the result of the comparison operation and the instructions placed on.

To better understand the operation, go to the next point, in which we discuss the "markers" mentioned above.

FLAGS

The value we get as a result of using the CMP instruction is saved in the so-called "flags". To be more precise, these are markers that are set in the appropriate positions, which then read other commands. For this reason, we will use the terms "flag" and "tag" interchangeably in the remainder of the book.

We have several types of conditional flags available, thanks to which the program can behave in various ways. They have their names and are marked with a short description, as below:

- | | |
|-----|-----------------|
| - C | - Carry flag |
| - V | - oVerflow flag |
| - Z | - Zero flag |
| - N | - Negative flag |
| - X | - eXtended flag |

Individual instructions can change their status, namely:

- "turn on" - set to 1,
- "turn off" - set to 0 (zero)

or reverse their current status. It all depends on the result of a particular comparison or the execution of an associated instruction. Please note that the tags allow you to separate additional bits from arithmetic and logic operations, thanks to which we can influence the program in a different way.

- Carry flag

The first type of flag is set in a situation when during the arithmetic operations the transfer from the most significant bit to the outside occurred (when adding) or the borrowing from the outside of the one to this bit (when subtracting).

To better understand this, take a look at the example below:

addi.b \$04,d0

Thanks to this line, we add a byte to data register D0. If the register will contain, for example, the number:

000000FE

the result of the operation will be:

102

In this situation, only 02 is saved in the D0 register, so its contents will be as follows:

00000002

The missing one will be saved in the transfer flag, because it does not "fit" in the byte. The Carry flag will therefore be set to 1.

If the result of the operation is different, for example 94, the contents of the register will look like this:

00000094

The same principle applies to the use of other instructions, for example LSR, about which we write, inter alia, in the point titled "Moving bits". It should be noted that every bit that will be removed from the register is transferred to the Carry flag.

- Overflow flag

This type of flag is set when the operation performed results in moving to the oldest bit or borrowing from this bit. Therefore, the result can not be saved in the bit being used. For example, if the following register value D0:

0000007D

you will add 04, i.e. enter such a line:

addi.b \$04, d0

The register will contain the value:

00000081

If we treat the byte as "signed" type, 7D value is positive, while 81 is negative. This change from positive to negative will set the Overflow marker, because the result of adding the two above values can not be negative.

The same principle applies "in the other direction", when the result of operations on two negative values would turn out to be positive. In other words, the excess marker indicates an invalid operation from an arithmetic point of view.

- Zero flag

It is the easiest to explain, because it indicates the fact that the result of the operation is the value zero. For example, if the contents of the D0 register are as follows:

0024000C

and you will use the following line:

subi.b \$0C, d0

In this situation, the contents of the registry will change to the following:

00240000

and the zero marker will be set, that is, it will get the value. If the byte is not zero, the flag will be "off", so it will contain zero. It can be confusing, to make it easier to distinguish two operations - setting (or "turning on") the zero flag and the result of the operation, which can be zero.

- Negative flag

Another flag, whose status depends on the sign of the obtained value. It indicates that the result of the operation is negative. For example, if register D0 contains the following value:

00049044

performance of the following line:

addi.w \$0100,d0

will result:

00049144

The word in the register is negative, therefore the character marker will be set. If the result were positive, the flag would be "disabled". As in the case of the zero mark, also here the difference between the result of the operation and the state of the marker should be distinguished.

- Extended flag

This type of flag is similar to the move marker, but its state depends on the "moved" value. The instructions can change both the Carry and Extended flags, they can also change the status of only one marker. In practice, the extension marker is an additional parameter that indicates the use of specific functions. Therefore, the importance of its condition depends on the operation being called. Practical information on this subject can be found, among others, in the section "Moving bits".

ADVANCED OPERATIONS

- Converting numbers into text strings

In many cases, you need to replace the number recording system, for example, to present them later on the screen. This is a quite complicated activity, as we have already written about. The matter can be simplified using text, that is, saving the values as a string.

Suppose that the D1 data register contains a long word, which should be converted to an 8-character ASCII string, that is, "plain" text. We will save it to a specific place in the memory so that it can be displayed later on the screen. The advantage of using hexadecimal instead of decimal in this case lies in the fact that to find a digit it is enough to read information about 4 bits (ie half byte), because it contains one hexadecimal digit. So we will use half of the byte in another register - with the symbol D2.

To get a sign that can be displayed, we must use the correct ASCII code. The 16-character codes that are used as hexadecimal digits are as follows:

0	1	2	3	4	5	6	7	8	9	A	B	C	D
E	F												
\$30	\$31	\$32	\$33	\$34	\$35	\$36	\$37	\$38	\$39	\$41	\$42	\$43	
\$44	\$45	\$46											

To convert the numbers 0-9, just add the value \$30. For letters A-F, which correspond to 10-15, add \$37.

Our program must determine the interval between 0-9 and A-F values and add the mentioned values, ie \$30 or \$37.

An example of a program that realizes these functions might look like this:

```
and    #$0f, d2
add    #$30, d2
cmp    #$3a, d2
bcs    ok
add    #7, d2
rts
```

This procedure converts the "half byte" in register D2 to an ASCII character that corresponds to the hexadecimal value. After adding \$30, we compare the value obtained and - if we have a number - we finish work. Otherwise, add \$7 to get \$37 together.

To convert the entire byte, we must call the procedure twice, which we will show in the next example. We now assume that the A0 register contains the address of the buffer in which characters will be inserted, and D1 contains the converted byte. That is why we will write a longer and more complicated program:

```
lea    buffer, a0
move   #$4a, d1
bsr    byte
rts
byte:
move   d1, d2
lsr    #4, d2
bsr    nibble
move.b d2, (a0) +
```

```

        move    d1, d2
        bsr     nibble
        move.b  d2, (a0)+
        rts
nibble:
        and     #$0f, d2
        add     #$30, d2
        cmp     #$3a, d2
        bcs     ok
        add     #7, d2
ok:
        rts
buffer:
        blk.b   9, 0

        end

```

How it works? First, it moves the value that we want to convert to the D2 register. It then moves the register four times "right" to move the top nibble to the four lower bits. After calling the subroutine, we use the line:

```
move.b d2, (a0)+
```

to place the upper part in the buffer. Then the original byte is re-entered into D2 and it is converted. This gives us the bottom four bytes as an ASCII character in the D2 register.

We place it in the next byte of the buffer. The whole closes the zero byte, which is usually required before the procedures displaying information on the screen. We will talk about it later.

Now let's deal with the "word" conversion. When converting a long word, we need to be sure that we will deal with the "half byte" in the right order. Before the first call to the "nibble" procedure, we must move the upper batch to the lower four bits of the long word without losing any data. In this case, the LSR command is not the best. It is better to use bits that rotate bits, such as ROR or ROL. You can read more about them in the section titled "Moving bits".

If we move the original long word in register D1 four times to the left, the four upper bits will be placed in the lower four bits. Now we can use our "nibble" procedure and then insert the obtained ASCII character into the buffer. We have to repeat it eight times, so that the whole long word will be converted. On the other hand, the D1 register will look exactly the same as before the entire process. The program realizing these functions is as follows:

```
hexlong:
    lea        buffer,a0
    move.l     #$12345678,d1
    move       #7,d3
loop:
    rol.l      #4,d1
    move       d1,d2
    bsr        nibble
    move.b     d2,(a0)+
    dbra       d3,loop
    rts
nibble:
    and        #$0f,d2
    add        #$30,d2
    cmp        #$3a,d2
    bcs        ok
```

```
        add      #7, d2
ok:      rts
        buffer:
        blk.b    9, 0

        end
```

Now let's look at another problem, namely the conversion of a four-digit decimal number. This is more complicated because you can not group bits to create single digits. You must use a different method.

Let's look at how the decimal number is constructed. In the four-digit issue, the highest place is in the place of a thousand, and the next is hundreds. If we have a saved value in the register and divide it by 1000, we will get a result which will show the highest position in the decimal number.

As we know, the DIV command not only gives us the result of the division, but also gives us the rest. That's why you just need to divide it by 100 to find more positions denoting hundreds and tens, then individual units.

Below we present a program that implements the above operations:

```
main:
        lea      buffer, a0
        move     #1234, d1
        jsr      deci_4
        illegal
```

```
deci_4:
    divu    #1000,d1
    bsr     digit

    divu    #100,d1
    bsr     digit

    divu    #10,d1
    bsr     digit

digit:
    add     #$30,d1
    move.b  d1,(a0)+
    clr     d1
    swap    d1
    rts

buffer:blk.b 5,0
end
```

The result will be stored in data register D1.

- Converting text strings to numbers

In a string, each hexadecimal digit represents a half byte. Just write a program that will reverse what the program did during the hexadecimal conversion. We wrote about it earlier.

First, try converting one digit. We will pass it to the pointer in address register A0. We want the binary value to return to the D0 data register. The program may look like:

```
        move.l    #string,a0
        jsr      nibblein
        nop

nibblein:
        clr.l     d0
        move.b    (a0)+,d0
        sub       #'A',d0
        bcc       ischar

        add       #7,d0
ischar:
        add       #10,d0
        rts

string:
        dc.b     'B',0

        end
```

Now let's trace how it works. Register A0 indicates a memory location that contains the character "B" which is represented by the ASCII value \$42. This number is loaded into the D0 register as soon as it is cleared. After deducting \$41, we get 1. Before returning from the subroutine, we add 10 to get the correct value of \$B. If there is a digit in the buffer, the subtraction will cause the register to become negative. That's why the Carry flag is set.

This program has one drawback. If a character is entered that is not a hexadecimal digit, we will get an incorrect result. To eliminate it, we have to deal with more complicated examples, which we will do later.

You can also convert a character string to a decimal number. To do this, you must use a similar method as before:

```
decin:
    clr.l    d1
    move.l   #string,a0
    jsr      decinloop
    nop

decinloop:
    bsr      digitin
    cmp      #10,d0
    bcc      decinok
    mulu     #10,d1
    add      d0,d1
    bra      decinloop

decinok:
    rts
```

```
digitin:
    clr.l    d0
    move.b   (a0)+,d0
    sub      #'0',d0
    rts

string:
    dc.b     '123456'

end
```

Please note that this program can convert numbers only to 655350. This is due to the fact that the MULU command can only multiply 16-bit words.

- Moving bits

In "Converting numbers to text strings" used commands for changing the position of bits in registers. These are the instructions about the names: ROR and ROL. The first one "shifts" the bits to the right (Right), the second - to the left (the Left).

Contrary to appearances, both commands are easy to understand, perhaps even more than the previous ones. For example, if in register D0 we will have the following contents of the byte written:

1100 0100

To invoke the left-shift operation, we execute the instruction as below:

rol.b #\$01,d0

Now all the digits will change their positions. Bits that will be in the extreme position will be saved on the "right" side. Therefore, the Carry marker will be set, which we write about in the point titled "Tags or flags". In short, thanks to Carry, you know how the shift was triggered.

You can move bits not only for one "position", but also for many, for example 2 or 4. These are examples of such lines:

rol.b #\$02,d0

and

rol.l #\$04,d2

Taking into account the previous contents of the register, after the first command has been executed, we will receive the following entry in the registry:

1100 1010

You can also perform a bit shift using a data register, for example:

rol.l d0,d1

In the case of moving to the right, i.e. the ROR instruction, the same rules apply. As before, all bits that go out of range on the right appear on the left.

Both commands have their limitations. You can not specify any number as an argument, because the maximum number of bits for which the data can be moved in the register is specified. It depends on the selected size.

If you use the "byte" (that is, you enter the instruction with the tip ".b"), the maximum offset is 08 bits. When you want to use "word" or "long word" appropriate values up to 10 bits and 1 F.

More about the use of different sizes of data we write in the section titled "Numbers".

Please note that the above restrictions are theoretical, because if you use larger - do not cause a program error. However, as a result you will get a zero value record. Therefore, you do not have to worry that your program will not

work, but if you want to do the correct calculation, you must follow the above rules.

With the bit-shifting operation, two other instructions are also called LSR and LSL. They perform a so-called "logical" shift, which differs from previous ROR and ROL commands. Now bits in the extreme positions will not appear "on the other side", but will be lost. At the same time, the following tags will be set: transfers and extensions. They will obtain the value of the least significant bit, while in the place of the most significant bit the zero value will appear.

The maximum offset size that can be performed in one instruction is 08 bits. If you want to call an offset of more than 08 bits, you can use the same instruction repeatedly. This may look like this:

```
lsl.l #$08,d0
```

and soon after:

```
lsl.l #$04,d0
```

The first instruction will move the bits to the left by 08 "position", while the second - to the left by an additional 4 bits. Together, this causes a 0C shifted in hexadecimal.

If we want to make a left shift by only 1 bit, we can enter the same command without giving any number, for example:

```
lsl.w d0
```

In practice, the statement will be executed in the following form:

```
lsl.w #$01,d0
```

However, the whole entry is faster and takes less memory space. Please note that both of the above forms are equivalent, and therefore work identically.

The LSR instruction works similarly, but it does the logical shift of the "right" bits. The principle of operation is the same, but take into account that you will get a completely different result.

- Attaching source files

You can attach other "source" files containing variables and other elements that can be used without the need to physically include them in the editor. To be included, however, first enter the line containing the INCLUDE command and the name of the corresponding file. E.g:

```
include my/macro.i
```

Please note that before the file name with the extension ".i" we have entered the directory where it was saved. Of course, this solution assumes that all files that make up your program are in the same current directory. This is the easiest way that does not require copying files and moving to subsequent directories.

In addition, you can specify a different directory where the files are saved. Just use the INCDIR command next to giving the name of the directory, i.e. for example:

```
incdir „Work:mime3/“
```

After calling the INCLUDE line, you can use all the data contained in the file as if you entered it in the program. The extension ".i" is typical for this type of file and you should stick to it. Due to the name of the instruction, they are also referred to as "Include files" or simply "Includes".

You can also add more than one file ", .i" - just enter multiple lines with the INCLUDE command. For example, as in the illustration:

```
include exec/memory.i
include dos/dos.i

include own/jvamacros.i
include libraries/execxref.i
include libraries/dosxref.i
```

These files can also contain other "source" files, but only up to five items. This function helps you to write a program faster, because in "Include" files you can save frequently used "universal" data or those that are repeated in different places, for example when your program divided into several shorter files. We are also writing about this under the heading "Working on multiple files".

OPERATING SYSTEM

Now we can deal with more complex programs. If we want to do a series of typical activities, we do not necessarily have to write them from scratch. Often, we can use the functions of the operating system through so-called libraries. Many of them can be found on the system disk in the "Libs" directory, others are stored in the ROM memory, ie the Kickstart system.

Here is a brief overview of the basic items:

- Exec.library

This library is needed to load other libraries. It is already in memory and does not have to be loaded. It supports basic functions such as memory reservation and input / output operations.

- Dos.library

It contains all the functions of normal input / output operations, for example disk access.

- Intuition.library

Used to work with screens, windows, menus and other elements of the graphical interface.

- Diskfont.library

Used to work with fonts saved on disk.

- **Graphics.library**

It contains functions controlling the Blitter system, it is used to perform basic graphical functions.

- Icon.library

Used when creating and using icons on Workbench.

- Mathffp.library
- Mathieeedoubbas.library
- Mathtrans.library

Used for mathematical operations.

- Timer.library

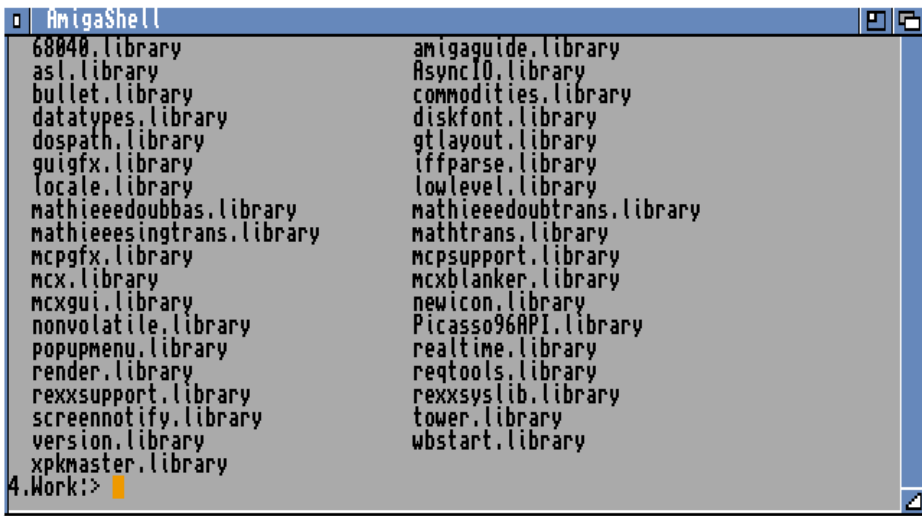
It contains the procedures used, for example, for programming exact intervals of operation of various parts of the program.

- Translator.Librar

It contains a function whose task is to convert text written to a computer "speech" (speech synthesizer).

As you can see, libraries contain procedures that are used every day when running other programs.

If you want to know the names of other libraries, look at the directory of the logical device "LIBS:" on your system drive. Here's how it can look:

A screenshot of an AmigaShell window. The title bar reads "AmigaShell". The window contains a list of system libraries arranged in two columns. The left column lists: 68040.library, asl.library, bullet.library, datatypes.library, dospath.library, guigfx.library, locale.library, mathieeedoubbas.library, mathieeesingtrans.library, mcpgfx.library, mcx.library, mcxgui.library, nonvolatile.library, popupmenu.library, render.library, rexxsupport.library, screennotify.library, version.library, and xpkmaster.library. The right column lists: amigaguide.library, AsyncIO.library, commodities.library, diskfont.library, gtlayout.library, iffpase.library, lowlevel.library, mathieeedoubtrans.library, mathtrans.library, mcpsupport.library, mcxblanker.library, newicon.library, Picasso96API.library, realtime.library, reqtools.library, rexxsyslib.library, tower.library, and wbstart.library. At the bottom left, the prompt "4.Work:>" is visible with a cursor.

```
AmigaShell
68040.library      amigaguide.library
asl.library        AsyncIO.library
bullet.library     commodities.library
datatypes.library  diskfont.library
dospath.library    gtlayout.library
guigfx.library     iffpase.library
locale.library     lowlevel.library
mathieeedoubbas.library
mathieeesingtrans.library
mcpgfx.library     mathieeedoubtrans.library
mcx.library        mathtrans.library
mcxgui.library     mcpsupport.library
nonvolatile.library
mcxblanker.library
newicon.library
Picasso96API.library
popupmenu.library  realtime.library
render.library     reqtools.library
rexxsupport.library
rexxsyslib.library
screennotify.library
tower.library
version.library    wbstart.library
xpkmaster.library

4.Work:>
```

We can use these functions, thanks to which the program can be created faster. In addition, you will be able to talk about it that it is compatible with the operating system, which increases compatibility between different hardware configurations.

However, before you can use any library, it must be loaded into memory. We will now tell you how to do it.

CALLING THE LIBRARIES

The use of libraries has the limitation that each must be loaded into memory in its entirety. This is necessary even if you want to use only one function. Remember that this takes time and takes up valuable computer memory. Therefore, you should always first consider what features you need and only then plan the use of specific libraries.

Special status has the "exec" library mentioned at the beginning, because its functions are available immediately after starting the computer. If you want to use a function from another library, the system must be informed and then the appropriate library must be called. If it has already been called, you must indicate where it is. Usually, all that is known about the library is its so-called base address, from which you can call the desired function.

The general scheme for calling the library is as follows:

```
move.l libBase, a6  
jsr     adres(a6)
```

The address of the corresponding function should be taken from the table in the Appendix at the end of the book. Note that all parameters must be loaded into the appropriate registers before calling the function.

Libraries in memory begin with many JMP commands that affect the algorithms in the library. To call a function, we must find the beginning of the jump table and call the specific function by going to the appropriate JMP instruction. This can be achieved by means of a so-called offset or - speaking in Polish - a displacement factor.

An unusual solution is the fact that we will start from the end of the jump table, not from the beginning. Therefore, we will use offsets with negative values. For example, to call "dos" libraries using the base address of the "exec" library, i.e. the following address:

\$000004

If you want to use a different library, we simply change the base address. Next, we need to enter the offset value (offset) for the function to be started. We open the library, so the function will be needed:

OpenLib

In addition, we must use a long word in memory to store the "dos" library address in it. You need to be sure that the name of the library will be written in lowercase, so together:

dos.library

otherwise you will not be able to open it. You should pay special attention to this, because omitting this simple rule will cause that the program will not work and we can devote many hours to the analysis of the seemingly correct code.

The finished procedure using the library named "dos" looks like this:

```
Execbase = 4  
OpenLib  = -408  
IoErr    = -132
```

```
init:
    move.l    Execbase,a6
    lea       dosname,a1
    moveq     #0,d0
    jsr       OpenLib(a6)
    move.l    d0,dosbase
    beq       error
    ...

error:
    move.l    dosbase,a6
    jsr       IoErr(a6)      ;
    move.l    d0,d5
    ...
    rts

dosname:
    dc.b      'dos.library',0,0

dosbase:
    blk.l     1

end
```

In this way, you can load the library so that you can continue to use it. In this way, we will gain access to all library functions. The parameters are placed in data registers D0 and D5, and then transferred to the function.

When an error occurs, the function does not work properly and in the data register D0 we usually get a zero value. This is another fact that you should remember.

Let us add that after the program is finished, all still "open" libraries should be closed before returning to the AmigaDOS or Workbench window. For this purpose, we will use the CloseLib function, which has the following offset value:

-414

Of course, it is in the "exec" library, like OpenLib. The only parameter that it needs to operate is the base address of the library to be closed. So, to close "dos.library", we do the following:

```
CloseLib = -414
...
move.l   Execbase, a6
move.l   dosbase, a1
jsr      CloseLib(a6)
```

- Initializing the program

So far, we've talked about the use of specific machine code functions. However, before the program is ready for work, you have to "activate" many things. One of the basic is access to memory. The program in which the user will enter the text must also be able to display data in the communication window. Therefore, depending on the function of your program, you will have to "turn on" one or more system libraries.

In the previous point, we started with calling the "dos" library as a good starting point. Now we will discuss further steps to make the program work properly in the operating system environment.

- Reserving the memory

There are several ways for the operating system to allocate a block of memory for use in the program. It is necessary that when working in multitasking one program does not overwrite the memory area used by another program.

Let's look at one of the typical functions called AllocMem. It is in the "exec" library and has an offset with the following value:

-\$c6

It reserves the memory area using the data register D0 to determine the size. As a result of its operation, the initial address of the memory area is returned to the registry. We can also use the word in register D1 to determine if the reserved memory area should be deleted or not.

In a ready-made program, the procedure using the AllocMem function looks as follows:

ExecBase = 4

AllocMem = -\$c6

```
...  
move.l    #number, d0  
move      #mode, a6  
move.l    ExecBase, a6  
  
jsr       AllocMem(a6)
```

```
move.l  d0, address
beq      error
...
```

The second way to reserve memory is to use the AllocAbs function whose offset value is:

\$CC

This function, unlike the AllocMem function, reserves a specific memory area. The D0 register contains the number of bytes that should be reserved. The address register A1 then has the desired start address. If it is not possible to reserve a memory area, the function returns a zero value to register D0.

Here is another example of usage:

```
ExecBase = 4
AllocAbs = -$cc

...
move.l  #number, d0
lea      address, a1
move.l  exectbase, a6
jsr      AllocAbs(a6)
tst.l    d0
beq      error
...
```

When the program does its work, it must return to the AmigaDOS or Workbench window.

Thus, he must return the reserved memory. This operation is supported by another FreeMem function with the following offset value:

-\$D2

It works similarly to AllocAbs but the number of bytes is placed in the data register D0, and the initial address of the memory area is placed in the address register A1. Remember that if you try to free up a memory area that was not previously reserved, you will crash or hang up.

The procedure for returning a reserved memory area is as follows:

xexBase = 4

FreeMem = -\$d2

```
...  
move.l    #number, d0  
lea       address, a1  
move.l    ExecBase, a6  
jsr       FreeMem(a6)  
tst.l     d0  
beq       error  
...
```

- Opening the window

Let's move to a slightly more complicated example. Opening a communication window can be useful for displaying a lot of useful information. You can do it in many ways, at this point we will present the simplest one. It will not work in every case, because it does not support all window buttons, but you will have the most important ones at your disposal, namely:

- closing button in the upper left corner,
- resize button in the bottom right corner,

To open the window, use the function from the "dos" library, so you must first open the correct library. We wrote about it earlier. It should look like this:

```
ExecBase = 4
OpenLib = -408
Open = -30

init:
    move.l    ExecBase, a6
    lea      dosname(pc), a1
    move.q    #0, d0
    jsr      OpenLib(a6)
    move.l    d0, dosbase
    beq      error
    ...
```

```
error:
    ...

OpenFile1:
    move.l    dosbase,a6
    jsr      Open(a6)
    tst.l    d0
    rts

dosname:
    dc.b      'dos.library',0,0

dosbase:
    blk.l    1
```

Please note that we call the "OpenFile1" subprogram, which must have specified parameters - stored in data registers D1 and D2. The first one will specify the file name, which must be terminated with a zero byte. The D2 register contains a parameter for the file opening function - the so-called "mode" in which the function should operate.

We can use two main values here:

- **1005** - opening an existing file for reading or writing data,
- **1006** - creating a new file - the old one is removed from the disk.

Therefore, in the long-term D2 register we have to pass the value 1005 or 1006. AmigaDOS in a similar way allows using input / output functions.

By default, a communication window is used which will open automatically. Just enter lines similar to the following:

MyOutputWindow:

```
dc.b 'CON:0/100/640/100/Window',0
```

Here we define the horizontal (X) and vertical coordinates (Y) of the upper left window, its size, and the name that should appear in the window's title line.

The whole scheme of the program will look like this

```
mode_old = 1005
```

```
lea      consolname(pc),a1
move.l   #mode_old,d0
bsr      openfile
beq      error
move.l   d0,Handle

rts
```

```
...
```

Handle:

```
dc.l 1
```

Please note that the determination of the operating mode takes place before opening the window. If the operation fails, the data register D0 will contain the zero.

When the window is opened, in the same register you will get the number of the so-called "handle", ie the number assigned to the communication window. It should be given in relation to each function that wants to use the window. In the above example, the handle was saved in a long word using the line after the label "HandleClip".

Mouse operation is supported by the Amiga operating system, so you do not have to worry about it.

An important function that uses the handle number is to close the window. It can still be found in the "dos" library under the most common name "Close". It has the offset value:

-36

and you must specify only one parameter - the number of the handle, which must be in data register D1. For example, like this:

Close = -36

```
...  
move.l  conhandle, d1  
move.l  dosbase, a6  
jsr     Close(a6)
```

As a result, the communication window will be closed.

If you open several windows in the same way, you will naturally get several windows, and thus several handle numbers. In this way, you can put multiple windows on the screen and support different functions if needed.

Handle numbers are independent, so we can operate on many windows or on a specific one, everything depends on the purpose of our program.

Here is another program scheme for opening and closing a simple window:

```
OpenLib      ==-30-378
closelib     ==-414
```

```
open         ==-30
close        ==-30-6
IoErr        ==-132
mode_old     = 1005
alloc_abs    ==-$cc
```

```
INCLUDE includes/Amiga.i
```

```
run:
```

```
    bsr      init
    bra      test
```

```
init:
```

```
    move.l   execbase, a6
    lea      dosname(pc), a1
    moveq    #0, d0
    jsr      openlib(a6)
    move.l   d0, dosbase
    beq      error
```

```

        lea      consolname(pc),a1
        move.l   #mode_old,d0
        bsr      openfile
        beq      error
        move.l   d0,conhandle

        rts

test:
        bra     qu

error:
        move.l   dosbase,a6
        jsr      IoErr(a6)
        move.l   d0,d5
        move.l   #-1,d7

qu:
        move.l   conhandle,d1
        move.l   dosbase,a6
        jsr      close(a6)

        move.l   dosbase,a1
        move.l   execbase,a6
        jsr      closelib(a6)

openfile:
        move.l   a1,d1
        move.l   d0,d2
        move.l   dosbase,a6
```

```
        jsr      open(a6)
        tst.l    d0
        rts

dosname:
        dc.b     'dos.library',0,0

dosbase:
        dc.l     0

consolname:
        dc.b     'CON:0/100/640/100/Test ',0

conhandle:
        dc.l     0

        end
```

You can open the window in the same way using the symbol:

RAW:

instead of "CON:".

All parameters and operations will remain the same and if you try both ways, you will not see any differences. Both windows will look the same and can be operated in the same way.

The difference appears when we enter data into the window. In the case of "RAW:" the user can not influence the content by pressing the cursor keys - they are ignored.

- Input/output operations

One of the most important functions of the program is data entry. For now, subprograms that perform specific functions have been presented, but now you have to add many other elements such as the ability to enter data from the keyboard. No less important are disk operations, without which the program will not be able to work effectively.

In order to prepare the input / output operations (ie input and output), appropriate libraries should be called up, and memory should be reserved. Operations of this type are associated with devices whose operation is carried out through the functions of the operating system. The system processes data or sends properly processed data to the device.

Let's start with the most important operation, that is data entry, to which the user of your program will usually use the keyboard.

- Keyboard support

Data from the keyboard can be read in a very simple way. All you need to do is open the communication window and read it and the information. For this purpose, the Read function has the following offset value:

-42

It has three parameters. After using it in register D1, you will receive a handle number so that you can handle reading data. The D2 register contains the address, and in the D3 register we find the number of bytes to be read.

Below is a subroutine that reads a specific number of characters entered from the keyboard, and then places the data in the buffer.

```
read      = -42  
  
          ...  
  
getchr:  
          move.l    #Bufor, d2  
          move.l    dosbase, a6  
          move.l    Handle, d1  
          jsr      read(a6)  
          rts  
  
Buffer:  
          blk.b    80, 0
```

I remind you that the procedure is a subroutine, so after execution it will return to the main program. If more than the number of characters indicated in the D3 data register has been entered, the procedure after the "Buffer" line takes only the first characters. Of course, you can call the remaining characters for the second time, while the number of characters moved to the buffer is placed in the D0 register.

The same algorithm can be used to read more of the entered characters. Here is an example:

```
move    #80,d3
bsr     readchr
lea     inline,a0
clr.b   0(a0,d0)
bsr     pmsg
```

This fragment should be made after opening the communication window. Next, of course, the window should be closed so that the program can be properly terminated. After launching, a cursor will appear in the upper left corner of the window. Enter the text and press ENTER. The text string should be displayed again on the screen.

Using a similar procedure, you can check the input of each individual character. This is especially useful when the program has to perform an important function and the user must confirm it with a single key.

In the "dos" library there is the WaitForCh function, which waits a specified time for pressing the key and returns the value zero if no key was pressed.

If it does not, we'll get -1 (or \$FFFFFFFF). This function requires two parameters. In the D1 data register, we must write the number of the handle from which the sign should be read. The D2 register is a place to specify the time to wait for the key to be pressed.

For example, to wait one second to press a key, you can use the following procedure:

WaitForCh=-30-174

...

scankey:

```
move.l  conhandle,d1
move.l  #1000000,d2
move.l  dosbase,a6
jsr     waitforch(a6)
tst.l   d0
rts
```

Let's now look at the complete program that will open and close the window, enter text and display it in the window:

```
openlib    =-30-378
closelib   =-414
open        =-30
close       =-30-6
read        =-42
write       =-48
```

```
IoErr      =-132
mode_old   =1005
alloc_abs  =-$cc
```

```
INCLUDE include/Amiga.i
```

```
run:
```

```
    bsr      init
    bsr      test
    nop
    bra      qu
```

```
test:
```

```
    move.l   #mytext,d0
    bsr      pmsg
    bsr      pcrlf
    bsr      pcrlf
    move.l   #80,d3
    bsr      getchr
    bsr      pmsg

    rts
```

```
init:
```

```
    move.l   execbase,a6
    lea      dosname(pc),a1
    moveq     #0,d0
    jsr      openlib
    move.l   d0,dosbase
```

```

        beq      error

        lea      consolname(pc),a1
        move.l   #mode_old,d0
        bsr      openfile
        beq      error
        move.l   d0,conhandle

        rts

pmsg:
        movem.l  d0-d7/a0-a6,-(sp)
        move.l   d0,a0
        move.l   a0,d2
        clr.l    d3

ploop:
        tst.b    (a0)+
        beq      pmsg2
        addq.l   #1,d3
        bra      ploop

pmsg2:
        move.l   conhandle,d1
        move.l   dosbase,a6
        jsr      write(a6)
        movem.l  (sp)+,d0-d7/a0-a6
        rts

```

```
pCrLf:
    move    #10,d0
    bsr     pchar
    move    #13,d0

pchar:
    movem.l d0-d7/a0-a6,-(sp)
    move.l  conhandle,d1

pch1:
    lea     outline,a1
    move.b  d0,(a1)
    move.l  a1,d2
    move.l  #1,d3
    move.l  dosbase,a6
    jsr     write(a6)
    movem.l (sp)+,d0-d7/a0-a6
    rts

getchr:
    move.l  #1,d3
    move.l  conhandle,d1
    lea     inbuff,a1
    move.l  a1,d2
    move.l  dosbase,a6
    jsr     read(a6)
    clr.l   d0
    move.b  inbuff,d0
    rts
```

error:

```
    move.l    dosbase,a6
    jsr      IoErr(a6)
    move.l    d0,d5

    move.l    #-1,d7
```

qu:

```
    move.l    conhandle,d1
    move.l    dosbase,a6
    jsr      close(a6)

    move.l    dosbase,a1
    move.l    execbase,a6 jsr
```

openfile:

```
    move.l    a1,d1
    move.l    d0,d2
    move.l    dosbase,a6
    jsr      open(a6)
    tst.l     d0
    rts
```

dosname:

```
    dc.b     'dos.library',0,0
```

dosbase:

```
    dc.l     0
```

```
consolname:
    dc.b 'CON:0/100/640/100/Test',0

conhandle:
    dc.l 0

mytext:
    dc.b 'Witaj!',0

outline:
    dc.w 0

inbuff:
    blk.b 8

end
```

- Disk operations

The most important peripheral device for Amiga was the floppy disk drive from the very beginning. If the program is to run on an unmodified computer, it is best to focus on servicing this media. That is why we will now look at the operations of reading and writing data.

Please note that the Amiga system allows very easy use of storage media. There are many so-called file systems, or ways to save and read information. Many diskettes, especially games, were once saved without using the operating system, so the data could not be read in the usual way, for example on Workbench.

When you try to access such a medium, a message similar to the following appears on the screen:



For this reason, many people describe such stored floppy disks as "non-DOS". We can say that the authors created in this way their own ways of recording information. However, we, the rest of the book, will look at the possibilities of using the functions of the operating system.

- Reading files

To access the file, you must first "open" it, or start the transmission channel. This is done using the Open function, which is located in the "dos" library. We wrote about her earlier.

For starters, this function requires setting the operating mode. This is very important, because if the file is to be opened in order to read data from it, it must already be created. We have the mode number 1005 and 1006, as discussed in the section "Opening the window". If the data is to be read only, we use mode 1005. If the file does not exist, we must use the value 1006.

However, in this case, when you save the data, the old file will be automatically deleted and overwritten. To avoid this, first check whether the file with the specified name exists.

That is why we will start from the subroutine that opens the file. The file must be specified by a DC instruction that reserves memory to store the name. It must end with a zero byte. Now just switch the mode in the D2 data register.

Assume that the file name begins with the 'filename' label and that it is closed with a zero byte. All you need to do is switch the mode in the D2 register. The procedure sets the number of the handle and returns to the main program. This operation is the last one performed by the subroutine, therefore its state can be evaluated only after the return. If the operation ran without problems and the file is opened, the handle will get a non-zero value.

Here is a ready subprogram that accomplishes these functions:

```
open      ==-30
close     ==-36
mode_old  = 1005
mode_new  = 1006
```

```
...
```

```
openfile:
```

```
    move.l  dosbase, a6
    move.l  #PlikNazwa, d1
    jsr     open(a6)
    move.l  d0, filehd
    rts
```

```
closefile:
```

```
    move.l  dosbase, a6
    move.l  PlikUchwyt, d1
    jsr     close(a6)
    rts
```

```
PlikUchwyt:
```

```
    dc.l    0
```

```
PlikNazwa:
```

```
    dc.b    "filename", 0
```

- Saving files

Let's try to create a new file now. At the beginning we introduce the following lines:

```
move.l    #mode_new, d2
bsr       openfile
beq       error
```

You can enter the same file name as we used before - after the "File Name" label. After calling the procedure, a new file will be created on the disk. Please note that if a file with the same name exists, it will be deleted.

We should save the data in the file, for example text. So we will use the following lines:

MyText :

```
dc.b      "This is test message", 0
```

MyTextEnd:

The label "MyTextEnd" is used so that you can calculate the number of bytes of data to the end of the entered string. Now we will use the Write function having an offset:

-48

Three parameters are required this time. In the D1 data register, we must have a file handle that we received as a result of the Open function. Register

with the designation D2 will now contain an address indicating the data that should be saved. In the D3 register - as before - we place the number of bytes to be written.

In order to obtain information to be recorded in the D2 and D3 registers, we must write another part of the program, as below:

```
Write  ==48  
  
    ...  
  
    writedata:  
        move.l  dosbase, a6  
        move.l  filehd, d1  
        jsr      write(a6)  
        rts
```

After opening the file, you can call the subroutine from the main program using the following lines:

```
    move.l  #text, d2  
    move.l  #textend-text, d3  
    bsr      writedata
```

Then close the file using a part similar to the following code:

```
    bsr      closefile  
    bra      end
```

After starting the program, the directory should contain a new file with the following name:

testfile

Due to the fact that we write "pure" text, the length of the file will coincide with the length of the entered text under the label "Text".

The next step may be to read the new file to make sure it contains the right data. To do this, we must use the Read function, which requires the same parameters as the Write. You can also use the aforementioned parameters by specifying the number of bytes to read only part of the file. If you specify a larger number than the file contains, all content will be loaded.

The number of bytes read will be placed in data register D0. However, you must reserve a memory location that will be sufficient to save the data read from the file. We can do it using the following line:

Buffer:

```
blk.b 100
```

In this case, we need only 100 bytes, but if you want to load another file, you can have much higher requirements. Here is a subroutine that reads the entire contents of an open file and places it in the buffer - data register D2:

```
read  = -42
```

```
...
```

```
readdata:
```

```
move.l  dosbase, a6  
move.l  filehd, d1  
move.l  #$ffffff, d3
```

```
jsr    read(a6)
rts
```

To now load the file into the buffer, use this procedure to load the file into "Buffer", we use the following program - outside the subroutine:

```
move,1  #mode_old,d2
bsr     openfile
beq     error
move.l  #field,d2
bsr     readdata
move.l  d0,d6
bsr     closefile
bra     end
```

After starting the program, in the D6 data register we will find additionally the number of bytes read from the file.

- Deleting files

Some files for use are no longer needed and should be deleted. For this purpose, we will use another function `DeleteFile`, which in the library "dos" for the following offset value:

-72

This time only one parameter is needed, i.e. the name of the file to be deleted. You must transfer it to the D1 data register. Remember that the file name must still be terminated with a zero byte.

The sample program that removes the file looks like this:

```
deletefile  =-72
...

move.l    dosbase, a6
move.l    #filename, d1
jsr      deletefile(a6)
```

- Renaming files

You will not always want to delete old files and save new ones. In many cases, you only need to change the name, for example to back up your data. We have another function from the "dos" library called Rename. Now the offset value is:

-78

In the D1 register, the first parameter must be found, i.e. the old file name, in the data register D2 - the new name, that is the second parameter of the function. For example, to change the file name from "file.txt" to "file.bak" use the following scheme:

rename =-78

...

```
move.l  dosbase, a6
move.l  #oldname, d1
move.l  #newname, d2
jsr     rename(a6)
...
```

oldname:

```
dc.b  "plik.txt", 0
```

newname:

```
dc.b  "plik.bak", 0
```

- Reading the directory using AmigaDOS commands

If you write a program that performs file operations, you will have to read the disk directory from time to time. There are several ways to do this. First, let's use the simplest method. It does not require a lot of work, because you can use the DIR or LIST commands located in the "C" directory on the system disk.

All we need to do this is to call the program as if we were running into the AmigaDOS window. This is provided by the function called "Execute", which has an offset with the following value:

-222

It requires three parameters. In the D1 data register, we must write the name of the command to be executed, once again ending it with a zero byte. Of course, this must be the same command as in the AmigaDOS window, otherwise the function will not work. In the D2 register, you can specify the text file handle that will be executed just like the AmigaDOS scripts are called - for example, the start sequence is saved in the system directory "S".

The D3 register will now contain the output file designation, but if we store the zero value in it, the data will be displayed in the standard AmigaDOS window.

To check the above information, use the following subroutine outline:

```
execute  = -222
          . . .
```

```
dir:
    move.l    dosbase, a6
    move.l    #command, d1
    clr.l     d2
    move.l    conhandle, d3
    jsr       execute(a6)
    rts

command:
    dc.b      "dir", 0
```

Of course, the "dos" library and window must be opened earlier, just as we discussed in the previous chapter. A similar program will work with other commands from the "C" directory, for example LIST. Remember that AmigaDOS commands can be used only if the appropriate files are on the system, which is a limitation. On the other hand, you use the system function directly, so if the user exchanges command files, it can automatically affect the operation of your program. In addition, it is short and uncomplicated.

Here is a ready program that calls the system DIR command:

```
openlib      =-408
closelib     =-414
open         =-30
close        =-36
execute      =-222
IoErr        =-132
mode_old     = 1005
alloc_abs    =-$cc
```

```
INCLUDE includes/Amiga.i
```

```
run:
```

```
    bsr    init
    bra    test
```

```
init:
```

```
    move.l  execbase, a6
    lea     dosname(pc), a1
    moveq   #0, d0
    jsr     openlib(a6)
    move.l  d0, dosbase
    beq     error

    lea     consolname(pc), a1
    move.l  #mode_old, d0
    bsr     openfile
    beq     error
    move.l  d0, conhandle

    rts
```

```
test:
```

```
    bsr    dir
    bra    qu
```

```
dir:
```

```
    move.l  dosbase, a6
    move.l  #command, d1
    clr.l   d2
```

```

        move.l  conhandle,d3
        jsr     execute(a6)
        rts

error:
        move.l  dosbase,a6
        jsr     IoErr(a6)
        move.l  d0,d5

        move.l  #-1,d7

qu:
        move.l  conhandle,d1
        move.l  dosbase,a6
        jsr     close(a6)

        move.l  dosbase,a1
        move.l  execbase,a6
        jsr     closelib(a6)

openfile:
        move.l  a1,d1
        move.l  d0,d2
        move.l  dosbase,a6
        jsr     open(a6)
        tst.l   d0
        rts

dosname:
        dc.b   'dos.library',0,0

```

dosbase:

dc.l 0

consolname:

dc.b 'CON:0/100/640/100/Test',0

conhandle:

dc.l 0

command:

dc.b "dir",0

end

- Reading the directory without using AmigaDOS commands

The functions presented in the previous item can be implemented without using commands from the system catalog "C. To do this, you must write your own procedure to read and display the disk directory.

To achieve this we will use a function called Lock. It requires two parameters. In the D1 register there must be a text containing the name of the directory that we want to read. This must be the same as when entering the access path, i.e. for example:

DFO:

for the main directory of the internal floppy disk drive. Register D2 this time will contain the designation of operating modes - reading or writing data. In the first case, we will use the Read mode.

The Lock function has the offset value:

-84

If the result is a zero value in data register D0, it means that the operation failed. Otherwise, we already know that the given access path is readable. Please note that in the same way you can check whether the file is saved to the disk.

The next necessary function is Examine. We will use it to find a file on the disk. As a result, we will receive a number of information such as below:

-
- file name,
 - file size,
 - file creation date.

To use them you need to reserve a block of memory and put the block start address in register D2 before calling the Examine function. The above-mentioned information together is referred to as the file information block, i.e. FileInfoBlock. It has a total length of 260 bytes. The name starts with the 9th byte and ends with a zero byte, which makes it easy to display.

Remember that the data provided by the Examine function, including in the event, will not affect a specific file, but a disk. Therefore, the name in FileInfoBlock is the name of the disk. The function sends the status information of the requested position back to the data register D0. We've called the Lock function before, so we do not have to worry about it now - we know in advance that the directory is readable.

Now we can go to the ExNext function, which is intended for reading individual files from the catalog. It searches for the next item each time it is called. The function requires the same parameters as Examine. If there are no more entries in the directory, ExNext stores the value zero in data register D0.

To find out that the directory "has finished", you still have to call the IoErr function, all the time from the "dos" library. It does not require any parameters and returns the state of the last operation that was performed in register D0. After the last call, we get the following value:

which means no more items in the catalog.

Here is the full procedure of reading the disk directory in the internal floppy disk drive "DFO:" and displaying its contents in the communication window.

openlib	==30-378
closelib	==414
exbase	=4
open	==30
close	==30-6
read	==30-12
write	==30-18
myinput	==30-24
output	==30-30
currrdir	==30-96
lock	==30-54
examine	==30-72
exnext	==30-78
exit	==30-114
IoErr	==30-102
waitforch	==30-174
mode	= 0
mode_old	= 1005
mode_new	= 1006
alloc_abs	== \$cc
free_mem	== \$d2

```
INCLUDE includes/Amiga.i
```

```
run:
```

```
    bsr    init
    bra    test
```

```
init:
```

```
    move.l  exbase, a6
    lea     dosname(pc), a1
    moveq   #0, d0
    jsr     openlib(a6)
    move.l  do, dosbase
    beq     error
```

```
    lea     consolname(pc), a1
    move.l  #mode_old, d0
    bsr     openfile
    beq     error
    moveq   d0, conhandle
```

```
    rts
```

```
test:
```

```
    move.l  #mytext, d0
    bsr     pmsg
```

```
    move.l  dosbase, a6
    move.l  #name, d1
    move.l  #-2, d2
```

```
        jsr      lock(a6)
        move.l   d0,d5
        tst.l    d0
        beq      error
        move.l   d0,locksav

        move.l   dosbase,a6
        move.l   locksav,d1
        move.l   #fileinfo,d2
        jsr      examine(a6)
        move.l   d0,d6
        tst.l    d0
        beq      error

loop:
        move.l   dosbase,a6
        move.l   locksav,d1
        move.l   #fileinfo,d2
        jsr      exnext(a6)
        tst.l    d0
        beq      error

        move.l   #fileinfo+8,d0
        bsr      pmsg
        bsr      pcrlf

bra      loop

error:
        move.l   dosbase,a6
```

```
jsr      ioerr(a6)
move.l   d0,d6
```

```
move.l   #presskey,d0
bsr      pmsg
bsr      getch
move.l   #-1,d7
```

qu:

```
move.l   conhandle,d1
move.l   dosbase,a6
jsr      close(a6)
```

```
move.l   dosbase,a1
move.l   exbase,a6
jsr      closelib(a6)
```

openfile:

```
move.l   a1,d1
move.l   d0,d2
move.l   dosbase,a6
jsr      open(a6)
tst.l    d0
rts
```

pmsg:

```
movem.l  d0-d7/a0-a6,-(sp)
move.l   d0,a0
move.l   a0,d2
clr.l    d3
```

```
mess1:
    tst.b    (a0)+
    beq      mess2
    addq.l   #1,d3
    bra      mess1

mess2:
    move.l   conhandle,d1
    move.l   dosbase,a6
    jsr      write(a6)
    movem.l  (sp)+,d0-d7/a0-a6
    rts

pcrlf:
    move     #10,d0
    bsr      pchar
    move     #13,d0

pchar:
    movem.l  d0-d7/a0-a6,-(sp)
    move.l   conhandle,d1

pch1:
    lea      chbuff,a1
    move.b   d0,(a1)
    move.l   a1,d2
    move.l   #1,d3
    move.l   dosbase,a6
```

```
jsr      write(a6)
movem.l  (sp)+,d0-d7/a0-a6
rts
```

scankey:

```
move.l   conhandle,d1
move.l   #500,d2
move.l   dosbase,a6
jsr      waitforch(a6)
tst.l    d0
rts
```

readin:

```
movem.l  d0-d7/a0-a6,-(sp)
lea      inline+2,a2
clr.l    (a2)
```

inplop:

```
bsr      getchr
cmp.b     #8,d0
beq      backspace
```

```
cmp.b     #127,d0
beq      backspace
bsr      pchar
cmp.b     #13,d0
beq      inputx
```

```

        move.b    d0, (a2)+
        bra       inplp

input:
        clr.b     (a2)+
        sub.l     #inline, a2
        move      a2, inline
        movem.l   (sp)+, d0-d7/a0-a6
        rts

backspace:
        cmp.l     #inline, a2
        beq       inplp
        move.b    #8, d0
        bsr       pchar
        move      #32, d0

        bsr       pchar
        move      #8, d0
        bsr       pchar
        clr.b     (a2)
        subq.l    #1, a2
        bra       inplp

getchr:
        move.l    #1, d3
        move.l    conhandle, d1
        lea       inbuff, a1
        move.l    a1, d2

```

```
        move.l  dosbase,a6
        jsr     read(a6)
        clr.l   d0
        move.b  inbuff,d0
        rts
```

```
mytext:
```

```
        dc.b  'Dir: DFO:',10,13,10,13,0,0
```

```
dosname:
```

```
        dc.b  'dos.library',0,0
```

```
presskey:
```

```
        dc.b  'Press ENTER',0
```

```
dosbase:
```

```
        dc.l  0
```

```
consolname:
```

```
        dc.b  'CON:0/100/640/100/Test',0
```

```
name:
```

```
        dc.b  'DFO:',0
```

```
locksav:
```

```
        dc.l  0
```

```
fileinfo:
```

```
        ds.l  20
```

```

conhandle:
    dc.l 0

inbuff:
    ds.b 8

inline:
    ds.b 180

chbuff:
    ds.b 82

end

```

The information block contains the following entries:

Offset	Name	Meaning
4	EntryType.L	Type of item (file or directory)
8	FileName	File name (108 bytes)
116	Protection.L	File protection status
120	EntryType.L	Type of item
124	Size.L	File size in bytes
128	NumBlocks.L	Number of blocks
132	Days.L	Creation date (day)
136	Minute.L	Creation time
140	Tick.L	Creation time
144	Comment	Comment (116 bytes)

If you want the program to also display the length of the file, you can read its length using the lines:

```
move.l fileinfo+124,d0
```

and then use the conversion procedure to get a decimal number. How to do it we wrote earlier.

- Selection windows

If you have only one floppy disk drive, you've probably seen a message like this one many times:



This type of windows is called "requester", because they have at least two options thanks to which we control the program. You can also create such windows in Assembler. Just use the "Intuition" library function called "AutoRequest". It has a shift value as below:

-348

It is responsible for displaying and managing selection windows. Of course, it also requires several parameters. In the register A0, an address containing the window structure should be placed, i.e. a piece of memory organized in a specific way. We will discuss this in more detail later. We put the address separately, of course using the label.

The A1 register should contain the next address, but the one with the saved message content, which will appear above the selection buttons. In

subsequent registers - with the symbols A2 and A3 - you should save the text which will be displayed on the left and right buttons.

In the data registers, however, information about the event should be placed, which should be called after selecting both buttons. We write them in the registers with the symbols D0 and D1. Use so-called IDCMP tags, which you can read more about in the item under "Event handling". The next two registers - D2 and D3 - should contain the width and height of the window.

Here is an example of a program that records the appropriate values:

```
autorequest ==348
    ...

request:
    move.l    windowhd,a0
    lea      btext,a1
    lea      ltext,a2
    lea      rtext,a3
    move.l    #0,d0
    move.l    #0,d1
    move.l    #180,d2
    move.l    #80,d3
    move.l    intbase,a6
    jsr      autorequest(a6)
    rts
```

Please note that in the first lines we have also indicated the so-called structure for the text, i.e. the parameters of the displayed message in the window. First, we define the colors of the text and the background.

The correct part of the program may look like this:

```
btext :  
    dc.b    2  
    dc.b    0
```

The next entries are text information, so they should be set in the appropriate positions - horizontal and vertical. Therefore, you should save the values defining the positions in relation to the upper left of the window. We do it like this:

```
    dc.w    10  
    dc.w    5
```

Next in the structure we have a pointer to the character set used. Write zero to use the standard set:

```
    dc.l    0
```

Then, you must enter the address of the text to be displayed. It must end with a zero byte:

```
    dc.l    text
```

Finally, we put a zero value, which means the end of the text that will be used in the window:

```
    dc.l    0
```

The whole part will of course be longer and may look like this:

btext:

dc.b 0,1
dc.b 0
align
dc.w 10,10
dc.l 0
dc.l bodytxt
dc.l 0

bodytxt:

dc.b "Message", 0

ltext:

dc.b 0,1
dc.b 0
align
dc.w 5,3
dc.l 0
dc.l lefttext
dc.l 0

lefttext:

dc.b "LEFT", 0
align

rtext:

dc.b 0,1
dc.b 0

```
dc.w    5,3
dc.l    0
dc.l    righttext
dc.l    0
```

```
righttext:
dc.b    "RIGHT",0
```

After calling the selection window, the address register D0 will contain information about which button was indicated by the user. If you get zero - it was the right button, if 1 appears - the user clicked the field on the left.

- Event handling

When you open a window, you usually want the program to respond appropriately to the user selecting buttons. To make this possible we need to get a signal, thanks to which it will be known that a certain action has been performed. In the operating system, such a situation is called an "event" and the signal simply - a message.

We can deal with various messages, which are defined by the so-called IDCMP tags. You do not have to worry about the meaning of this shortcut, but it is important to know what each message means. First, get acquainted with the list of the most important items:

Bit	Wartość	Nazwa	Znaczenie
1	\$000002	NEWSIZE	Zmiana rozmiaru okna
2	\$000004	REFRESHWINDOW	Odświeżenie zawartości okna
3	\$000008	MOUSEBUTTONS	Naciśnięty klawisz myszki
4	\$000010	MOUSEMOVE	Przesunięty wskaźnik myszki
8	\$000100	MENUPICK	Wybrana pozycja menu
9	\$000200	CLOSEWINDOW	Zamknięte okno
10	\$000400	RAWKEY	Naciśnięty klawisz
15	\$008000	DISKINSERTED	Włożona dyskietka
16	\$010000	DISKREMOVED	Wyjęta dyskietka
18	\$040000	ACTIVEWINDOW	Okno stało się aktywne
19	\$080000	INACTIVEWINDOW	Okno stało się nieaktywne

To get the right message, we'll use the "exec" library function called "GetMsg". It has the offset value, ie the offset:

-372

The source address of the event should be provided as the parameter. We can get it from the so-called user port, which contains entries specifying events that took place and related information. We will use the window structure address, which was read as a result of the window opening function.

The information will consist of several parts. Some are copies of the window parameters that we set. The most interesting data is in a long word that begins with 86 bytes. To read them, use the example below:

```
move.l  windowhd, a0
move.l  86(a0), a0
```

You can call the "GetMsg" function with the pointer in address register named A0:

```
GetMsg  = -372
...
move.l  windowhd, a0
move.l  86(a0), a0
move.l  execbase, a6
jsr     getmsg(a6)
```

This function returns the value in register D0, which is the address of the next structure - associated with the message provided in the operating system. If in the data register D0 we have a zero value it means that no event has taken place. Otherwise, we will receive information about what event has occurred. The information will be written in a long word beginning with 20 bytes. Bits have the same meaning as IDCMP tags, so you can easily read them.

To see if the event happened, you can use lines similar to the following:

```
move.l    d0,a0
move.l    20(a0),d6
tst.l     d0
bne       Koniec
```

For example, if the D6 register contains the following number:

\$00000200

this will mean that the user has chosen the button to close the program window.

HARDWARE REGISTERS

To use the various Amiga hardware features, you do not need to use system libraries. Instead, you can use the hardware registers. These are memory cells that are not in RAM or in ROM. They provide direct interfaces between the processor and peripheral devices. Each device has many hardware registers that the processor has access to, so you can control, for example, graphics or sound related functions.

You can find a list of registers in one of the appendices at the end of the book. We will use it in further considerations, as well as discuss their functions and use.

MOUSE AND JOYSTICK SUPPORT

To operate the mouse and joystick we have two hardware registers. They contain the status of these devices. The same port can be used to use both the mouse and the joystick, but they work completely differently. The joystick has four switches that change their positions during swing. The mouse, on the other hand, sends many short signals that denote horizontal and vertical movement.

Amiga has two mouse or joystick ports. Their status is saved at the following addresses:

- first port	\$DFF00A
- second port	\$DFF00C

To read the contents of the above registers we can use the following program:

```
test:
        jsr     run
        jmp     test
        nop

joy= $dff00a

run:
        move    joy, d6
        move    joy+2, d7
        rts
        end
```

We write the state of port registers to two data registers - symbols D6 and D7. If we move the mouse and check the contents of the registry, it turns out that it will change to D6. However, in this way we will not get the absolute position of the mouse pointer on the screen. We can easily see this by moving the mouse to the upper left corner, and then reading the value, restarting the program and trying to move the pointer to the left. The content of registers is always relative.

Therefore, the program should change the program as follows:

```
test:
        jsr      run
        jmp      test
        nop

joy= $dff00a

run:
        move     d7,d6
        move     joy,d7
        sub      d7,d6
        rts

        end
```

Now, if you move the mouse, the D6 register contains the difference between the old one and the new position, so you can read the vertical and horizontal position of the mouse relative to the previous one. In this way, we can find the relative movement of the mouse pointer.

Let's now handle the joystick. For this purpose, the program should change the address \$ DFF00A to:

\$DFF00C

The result will be saved in data register D7. If you throw the joystick up, you get the value:

\$FF00

If you release the joystick, you will get \$ 100 - the unit will be added. You will get the same effect if you move the joystick to the left, so after changing position to neutral, the unit will be subtracted.

To better understand these calculations, look at the list of individual movements and their impact on our program:

- up	\$FF00	upper byte -1
- down	\$FFFF	lower byte -1
- left	\$0100	upper byte +1
- right	\$0001	lower byte +1

This is the fastest way to read the joystick status. In this way, an external device that transmits readable and recalculated signals can be connected to the port and operated in a machine language program.

Now we have to check if the Fire button has been pressed. Button state is saved in byte 7 at the following address:

\$BFE001

If the bit is "on" (i.e., 1), the button has not been pressed. Please note that the joystick is usually connected to port number 2 in the Amiga. However, some games allow for the gameplay of two players and then the other joystick will work in port number 1 - instead of the mouse. The bit 6 indicates the status of the buttons for this port.

Therefore, in order for the program to perform a specific function, when the user presses the Fire button in the joystick connected to the port number 2, just use the following lines:

```
tst.b    $bfe001  
bpl     fire
```

The TST.B instruction tests the byte in the given address and sets the zero mark and character (i.e. negativity). We wrote about their importance earlier.

If the N flag is set, we know that bit 7 of the byte being tested is set. Remember that the bit is cleared (ie it has a value of 0) after pressing the Fire button. The BPL command causes a "jump" to another part of the program in a situation where the result of the comparison is non-negative.

Now we can combine all the information. Here's a complete program diagram for checking the joystick position and the status of the Fire button:

```
test :  
  
      jsr     run  
      tst.b    $bfe001
```

```
        bpl      fire
        jmp      test

joy = $dff00a

run:
        move     d7,d6
        move     joy,d7
        sub      d7,d6
        rts

fire:
        nop

        end
```

ENDING

The machine language is not the easiest one to understand not only but also to remember the elements necessary to run the program. Certainly many people will choose high-level languages, because long keywords and direct translation of the entered lines into subsequent operations are definitely more legible.

This is even more important if the user is just starting his programming adventure, which is the assumption of the cycle "Programming from scratch". However, I hope that this book will allow us to take the first step towards writing programs in natural language for our computers.

APPENDIX A

Library table

CLIST.LIBRARY

-\$001E -30	InitCLPool (cLPool,size) (A0,D0)
-\$0024 -36	AllocCList (cLPool) (A1)
-\$002A -42	FreeCList (cList) (A0)
-\$0030 -48	FlushCList (cList) (A0)
-\$0036 -54	SizeCList (cList) (A0)
-\$003C -60	PutCLChar (cList,byte) (A0,D0)
-\$0042 -66	GetCLChar (cList) (A0)
-\$0048 -72	UnGetCLChar (cList,byte) (A0,D0)
-\$004E -78	UpPutCLChar (cList) (A0)
-\$0054 -84	PutCLWord (cList,word) (A0,D0)
-\$005A -90	GetCLWord (cList) (A0)
-\$0060 -96	UnGetCLWord (cList,word) (A0,D0)
-\$0066 -102	UnPutCLWord (cList) (A0)
-\$006C -108	PutCLBuf (cList,buffer,length) (A0,A1,D1)
-\$0072 -114	GetCLBuf (cList,buffer,Length) (A0,A1,D1)
-\$0078 -120	MarkCList (cList,offset) (A0,D0)
-\$007E -126	IncrCLMark (cList) (A0)
-\$0084 -132	PeekCLMark (cList) (A0)
-\$008A -138	SplitCList (cList) (A0)
-\$0090 -144	CopyCList (cList) (A0)
-\$0096 -150	SubCList (cList,index,length) (A0,D0.D1)
-\$009C -156	ConcatCList (sourceCList,destCList) (A0,A1)

CONSOLE.LIBRARY

-\$002A -42	CDInputHandler (events,device) (A0,A1)
-\$0030 -48	RawKeyConvert (events,buffer,length,key) (A0,A1,D1,A2)

DISKFONT.LIBRARY

-\$001E -30	OpenDiskFont (textAttr) (A0)
-\$0024 -36	AvailFonts (buffer,bufBytes,flags) (A0,D0,D1)

DOS.LIBRARY

-\$001e	-30	Open (name,access mode)(d1,d2)
-\$0024	-36	Close (file)(d1)
-\$002a	-42	Read (file,buffer,length)(d1,d2,d3)
-\$0030	-48	Write (file,buffer,length)(d1,d2,d3)
-\$0036	-54	Input()
-\$003c	-60	Output()
-\$0042	-66	Seek(file,position,offset)(d1,d2,d3)
-\$0048	-72	DeleteFile (name)(d1)
-\$004e	-78	Rename(oldname,newname)(d1,d2)
-\$0054	-84	Lock(name,type)(d1,d2)
-\$005a	-90	UnLock(lock)(d1)
-\$0060	-96	DupLock(lock)(d1)
-\$0066	-102	Examine(lock,fileinfoblock)(d1,d2)
-\$006c	-108	ExNext(lock,fileinfoblock)(d1,d2)
-\$0072	-114	Info(lock,parameterblock)(d1,d2)
-\$0078	-120	CreateDir(name)(d1)
-\$007e	-126	CurrentDir(lock)(d1)
-\$0084	-132	IoErr()
-\$008a	-138	CreateProc(name,pri,seglist,stacksize) (d1,d2,d3,d4)
-\$0090	-144	Exit(returncode)(d1)
-\$0096	-150	LoadSeg(filename)(d1)
-\$009c	-156	UnLoadSeg(segment)(d1)
-\$00a2	-162	Getpacket(wait)(d1)
-\$00a8	-168	Queuepacket(packet)(d1)

-\$00ae	-174	DeviceProc(name)(d1)
-\$00be	-180	SetComment(name,comment)(d1,d2)
-\$ooba	-186	SetProtection(name,mask)(d1,d2)
-\$00c0	-192	DateStamp(date)(d1)
-\$00c6	-198	Delay(timeout)(d1)
-\$00cc	-204	WaitForChar(file,timeout)(d1,d2)
-\$00d2	-210	ParentDir(lock)(d1)
-\$00d8	-216	IsInteractive(file)(d1)
-\$00de	-222	Execute(string,file,file)(d1,d2,d3)

EXEC.LIBRARY

-\$001e	-30	Supervisor()
-\$0024	-36	ExitIntr()
-\$002a	-42	Schedule()
-\$0030	-48	Reschedule()
-\$0036	-54	Switch()
-\$003c	-60	Dispatch()
-\$0042	-66	Exception()
-\$0048	-72	InitCode(startclass,version)(d0,d1)
-\$004e	-78	InitStruct(inittable,memory,size)(a1,a2,d0)
-\$0054	-84	MakeLibrary(funcinit,structinit,libinit,datasize, codesize)(a0,a1,a2,d0,d1)
-\$005a	-90	MakeFunctions(target,functionarray,funcdispbase) (a0,a1,a2)
-\$0060	-96	FindResident(name)(a1)
-\$0066	-102	InitResident(resident,seglist)(a1,d1)
-\$006c	-108	Alert(alertnum,parameters)(d7,a5)
-\$0072	-114	Debug()
-\$0078	-120	Disable()
-\$007e	-126	Enable()
-\$0084	-132	Forbid()
-\$008a	-138	Permit()
-\$0090	-144	SetSR(newsrr,mask)(d0,d1)
-\$0096	-150	SuperState()
-\$009c	-156	UserState(sysstack)(d0)
-\$00a2	-162	setIntVector(intnumber,interrupt)(d0,d1)

-\$00a8	-168	AddIntServer(intnumber,interrupt)(d0,d1)
-\$00ae	-174	RemIntServer(intnumber,interrupt)(d0,d1)
-\$00b4	-180	Cause(interrupt)(a1)
-\$00ba	-186	Allocate(freelist,bytesize)(a0,d0)
-\$00c0	-192	Deallocate(freelist,memoryblock,bytesize) (a0,a1,d0)
-\$00c6	-198	AllocMem(bytesize,requirements)(d0,d1)
-\$00cc	-204	AlloAbs(bytesize,location)(d0,a1)
-\$00d2	-210	FreeMem(memoryblock,bytesize)(a1,d0)
-\$00d8	-216	AvailMem(requirements)(d1)
-\$00de	-222	AllocEntry(entry)(a0)
-\$00e4	-228	FreeEntry(entry)(a0)
-\$00ea	-234	Insert(list,node,pred)(a0,a1,a2)
-\$00f0	-240	AddHead(list,node)(a0,a1)
-\$00f6	-246	AddTail(list,node)(a0,a1)
-\$00fc	-252	Remove(node)(a1)
-\$0102	-258	RemHead(list)(a0)
-\$0108	-264	RemTail(list)(a0)
-\$010e	-270	Enqueue(list,node)(a0,a1)
-\$0114	-276	FindName(list,name)(a0,a1)
-\$011a	-282	AddTask(task,initpc,finalpc)(a1,a2,a3)
-\$0120	-288	RemTask(task)(a1)
-\$0126	-294	FindTask(name)(a1)
-\$012c	-300	SetTaskPri(task,priority)(a1,d0)
-\$0132	-306	SetSignal(newsignals,signalset)(d0,d1)
-\$0138	-312	SetExcept(newsignals,signalset)(d0,d1)
-\$013e	-318	Wait(signalset)(d0)

-\$0144	-324	Signal(task,signalset)(a1,d0)
-\$014a	-330	AllocSignal(signalnum)(d0)
-\$0150	-336	FreeSignal(signalnum)(d0)
-\$0156	-342	AllocTrap(trapnum)(d0)
-\$015c	-348	FreeTrap(trapnum)(d0)
-\$0162	-354	AddPort(port)(a1)
-\$0168	-360	RemPort(port)(a1)
-\$016e	-366	PutMsg(port,message)(a0,a1)
-\$0174	-372	GetMsg(port)(a0)
-\$017a	-378	ReplyMsg(message)(a1)
-\$0180	-384	WaitPort(port)(a0)
-\$0186	-390	FindPort(name)(a1)
-\$018c	-396	AddLibrary(library)(a1)
-\$0192	-402	RemLibrary(library)(a1)
-\$0198	-408	OldOpenLibrary(libname)(a1)
-\$019e	-414	CloseLibrary(library)(a1)
-\$01a4	-420	Setfunction(library,funcoffset,funcentry) (a1,a0,d0)
-\$01aa	-426	SumLibrary(library)(a1)
-\$01b0	-432	AddDevice(device)(a1)
-\$01b6	-438	RemDevice(device)(a1)
-\$01bc	-444	OpenDevice(devname,unit,iorequest,flags) (a0,d0,a1,d1)
-\$01c2	-450	CloseDevice(iorequest)(a1)
-\$01c8	-456	DoIO(iorequest)(a1)
-\$01ce	-462	SendIO(iorequest)(a1)
-\$01d4	-468	CheckIO(iorequest)(a1)

-\$01da	-474	WaitIO(iorequest)(a1)
-\$01e0	-480	AbortIO(iorequest)(a1)
-\$01e6	-486	AddResource(resource)(a1)
-\$01ec	-492	RemResource(resource)(a1)
-\$01f2	-498	OpenResource(resname,version)(a1,d0)
-\$01f8	-504	RawIOInit()
-\$01fe	-510	RawMayGetChar()
-\$0204	-516	RawPutChar(char)(d0)
-\$020a	-522	RawDoFmt()(a0,a1,a2,a3)
-\$0210	-528	GetCC()
-\$0216	-534	TypeOfMem(address)(a1)
-\$021c	-540	Procedure(semaport,bidmsg)(a0,a1)
-\$0222	-546	Vacate(semaport)(a0)
-\$0228	-552	OpenLibrary(libname,version)(a1,d0)

GRAPHICS.LIBRARY

-\$001e	-30	BltBitMap(scrbitmap,scrx,scry,destbitmap,destx, desty,sizex,sizey,minterm,mask,tempa) (a0,d0,d1,a1,d2,d3,d4,d5,d6,d7,a2)
-\$0024	-36	BltTemplate(source,scrx,scrmod,destrastport,destx, desty,sixex,sizey)(a0,d0,d1,a1,d2,d3,d4,d5)
-\$002a	-42	ClearEOL(rastport)(a1)
-\$0030	-48	ClearScreen(rastport)(a1)
-\$0036	-54	TextLength(rastport,string,count)(a1,a0,d0)
-\$003c	-60	Text(rastport,string,count)(a1,a0,d0)
-\$0042	-66	SetFont(rastportid,textfont)(a1,a0)
-\$0048	-72	OpenFont(textattr)(a0)
-\$004e	-78	CloseFont(textfont)(a1)
-\$0054	-84	AskSoftStyle(rastport)(a1)
-\$005a	-90	SetSoftStyle(rastport,style,enable)(a1,d0,d1)
-\$0060	-96	AddBob(bob,rastport)(a0,a1)
-\$0066	-102	AddVSprite(vsprite,rastport)(a0,a1)
-\$006c	-108	DoCollision(rastport)(a1)
-\$0072	-114	DrawGList(rastport,viewport)(a1,a0)
-\$0078	-120	InitGels(dummyhead,dummytail,gelsinfo)(a0,a1,a2)
-\$007e	-126	InitMasks(vsprite)(a0)
-\$0084	-132	RemIBob(bob,rastport,viewport)(a0,a1,a2)
-\$008a	-138	RemVSprite(vsprite)(a0)
-\$0090	-144	SetCollision(type,routine,gelsinfo)(d0,a0,a1)
-\$0096	-150	SortGList(rastport)(a1)
-\$009c	-156	AddAnimObj(obj,animationkey,rastport)(a0,a1,a2)

-\$00a2	-162	<code>Animate(animationkey,rastport)(a0,a1)</code>
-\$00a8	-168	<code>etGBuffers(animationobj,rastport,doublebuffer)(a0, a1,d0)</code>
-\$00ae	-174	<code>InitGMasks(animationobj)(a0)</code>
-\$00b4	-180	<code>GelsFuncE()</code>
-\$00ba	-186	<code>GelsFuncF()</code>
-\$00c0	-192	<code>LoadRGB4(viewport,colurs,count)(a0,a1,d0)</code>
-\$00c6	-198	<code>InitRastPort(rastport)(a1)</code>
-\$00cc	-204	<code>InitVPort(viewport)(a0)</code>
-\$00d2	-210	<code>MrgCop(view)(a1)</code>
-\$00D8	-216	<code>MakeVPort (view,viewPort) (A0,A1)</code>
-\$00DE	-222	<code>LoadView (view) (A1)</code>
-\$00E4	-228	<code>WaitBlit ()</code>
-\$00EA	-234	<code>SetRast (rastPort,color) (A1,D0)</code>
-\$00F0	-240	<code>Move (rastPort,x,y) (A1,D0,D1)</code>
-\$00F6	-246	<code>Draw (rastPort,x,y) (A1,D0,D1)</code>
-\$00FC	-252	<code>AreaMove (rastPort,x,y) (A1,D0,D1)</code>
-\$0102	-258	<code>AreaDraw (rastPort,x,y) (A1,D0,D1)</code>
-\$0108	-264	<code>AreaEnd (rastPort) (A1)</code>
-\$010E	-270	<code>WaitTOF ()</code>
-\$0114	-276	<code>QBlit (blit) (A1)</code>
-\$011A	-282	<code>InitArea (areaInfo,vectorTable,vectorTableSize) (A0,A1,D0)</code>
-\$0120	-288	<code>SetRGB4 (viewPort,index,r,g,b) (A0,D0,D1,D2,D3)</code>
-\$0126	-294	<code>QBSBlit (blit) (A1)</code>
-\$012C	-300	<code>BlitClear (memory,size,flags) (A1,D0,D1)</code>

-\$0132 -306	RectFill (rastPort,x1,y1,xu,yu) (A1,D0,D1,D2,D3)
-\$0138 -312	BltPattern (rastPort,ras,x1,y1,maxX,maxY, fillBytes) (A1,A0,D0,D1,D2,D3,D4)
-\$013E -318	ReadPixel (rastPort,x,y) (A1,D0,D1)
-\$0144 -324	WritePixel (rastPort,x,y) (A1,D0,D1)
-\$014A -330	Flood (rastPort,mode,x,y) (A1,D2,D0,D1)
-\$0150 -336	PolyDraw (rastPort,count,polyTable) (A1,D0,A0)
-\$0156 -342	SetAPen (rastPort,pen) (A1,D0)
-\$015C -348	SetBPen (rastPort,pen) (A1,D0)
-\$0162 -354	SetDrMd (rastPort,drawMode) (A1,D0)
-\$0168 -360	InitView (view) (A1)
-\$016E -366	CBump (copperList) (A1)
-\$0174 -372	CMove (copperList,destination,data) (A1,D0,D1)
-\$017A -378	CWait (copperList,x,y) (A1,D0,D10)
-\$0180 -384	VBeamPos ()
-\$0186 -390	InitBitMap (bitMap,depth,width,height) (A0,D0,D1,D2)
-\$018C -396	ScrollRaster (rastPort,dX,dY,minx,miny,maxx, maxy) (A1,D0,D1,D2,D3,D4,D5)
-\$0192 -402	WaitBOVP (viewPort) (A0)
-\$0198 -408	GetSprite (simpleSprite,num) (A0,D0)
-\$019E -414	FreeSprite (num) (D0)
-\$01A4 -420	ChangeSprite (vp,simpleSprite,data) (A0,A1,A2)
-\$01AA -426	MoveSprite (viewPort,simpleSprite,x,y) (A0,A1,D0,D1)
-\$01B0 -432	LockLayerRom (layer) (A5)

-\$01B6 -438	UnlockLayerRom (layer) (A5)
-\$01BC -444	SyncSBitMap (1) (A0)
-\$01C2 -450	CopySBitMap (11,12) (A0,A1)
-\$01C8 -456	OwnBlitter ()
-\$01CE -462	DisownBlitter ()
-\$01D4 -468	InitTmpRas (tmpras,buff,size) (A0,A1,D0)
-\$01DA -474	AskFont (rastPort,textAttr) (A1,A0)
-\$01E0 -480	AddFont (textFont) (A1)
-\$01E6 -486	RemFont (textFont) (A1)
-\$01EC -492	AllocRaster (width,height) (D0,D1)
-\$01F2 -498	FreeRaster (planePtr,width,height) (A0,D0,D1)
-\$01F8 -504	AndRectRegion (rgn,rect) (A0,A1)
-\$01FE -510	OrRectRegion (rgn,rect) (A0,A1)
-\$0204 -516	NewRegion ()
-\$0210 -528	ClearRegion (rgn) (A0)
-\$0216 -534	DisposeRegion (rgn) (A0)
-\$021C -540	FreeVPortCopLists (viewPort) (A0)
-\$0222 -546	FreeCopList (coplist) (A0)
-\$0228 -552	ClipBlit (srcrp,srcX,srcY,destrp,destX,destY, sizeX,sizeY,minterm (A0,D0,D1,A1,D2,D3,D4,D5,D6)
-\$022E -558	XorRectRegion (rgn,rect) (A0,A1)
-\$0234 -564	FreeCprList (cprlist) (A0)
-\$023A -570	GetColorMap (entries) (D0)
-\$0240 -576	FreeColorMap (colormap) (A0)
-\$0246 -582	GetRGB4 (colormap,entry) (A0,D0)
-\$024C -588	ScrollVPort (vp) (A0)

-\$0252 -594	UCopperListInit (copperlist,num) (A0,D0)
-\$0258 -600	FreeGBuffers (animationObj,rastPort, doubleBuffer) (A0,A1,D0)
-\$025E -606	BltBitMapRastPort (srcbm,srcx,srcy,destrp, destX,destY,sizeX,sizeY,minter) (A0,D0,D1,A1,D2,D3,D4,D5,D6)

ICON.LIBRARY

-\$001E -30	GetWBOobject (name) (A0)
-\$0024 -36	PutWBOobject (name,object) (A0,A1)
-\$002A -42	GetIcon (name,icon,freelist) (A0,A1,A2)
-\$0030 -48	PutIcon (name,icon) (A0,A1)
-\$0036 -54	FreeFreeList (freelist) (A0)
-\$003C -60	FreeWBOobject (WBOobject) (A0)
-\$0042 -66	AllocWBOobject ()
-\$0048 -72	AddFreeList (freelist,mem,size) (A0,A1,A2)
-\$004E -78	GetDiskObject (name) (A0)
-\$0054 -84	PutDiskObject (name,diskobj) (A0,A1)
-\$005A -90	FreeDiskObj (diskobj) (A0)
-\$0060 -96	FindToolType (toolTypeArray,typeName) (A0.A1)
-\$0066 -102	MatchToolValue (typeString,value) (A0,A1)
-\$006C -108	BumbRevision (newname,oldname) (A0,A1)

INTUITION.LIBRARY

-\$001E -30	OpenIntuition ()
-\$0024 -36	Intuition (ievent) (A0)
-\$002A -42	AddGadget (AddPtr,Gadget,Position) (A0,A1,D0)
-\$0030 -48	ClearDMRequest (Window) (A0)
-\$0036 -54	ClearMenuStrip (Window) (A0)
-\$003C -60	ClearPointer (Window) (A0)
-\$0042 -66	CloseScreen (Screen) (A0)
-\$0048 -72	CloseWindow (Window) (A0)
-\$004E -78	CloseWorkbench ()
-\$0054 -84	CurrentTime (Seconds,Micros) (A0,A1)
-\$005A -90	DisplayAlert (AlertNumber,String,Height) (D0,A0,D1)
-\$0060 -96	DiplayBeep (Screen) (A0)
-\$0066 -102	DoubleClick (sseconds,smicros,cseconds, cmicros) (D0,D1,D2,D3)
-\$006C -108	DrawBorder (Rport,Border,LeftOffset,TopOffset) (A0,A1,D0,D1)
-\$0072 -114	DrawImage (RPort,Image,LeftOffset,TopOffset) (A0,A1,D0,D1)
-\$0078 -120	EndRequest (requester>window) (A0,A1)
-\$007E -126	GetDefPref (preferences,size) (A0,D0)
-\$0084 -132	GetPrefs (preferences,size) (A0,D0)
-\$008A -138	InitRequester (req) (A0)
-\$0090 -144	ItemAddress (MenuStrip,MenuNumber) (A0,D0)
-\$0096 -150	ModifyIDCMP (Window,Flags) (A0,D0)

-\$009C -156	ModifyProp (Gadget,Ptr,Reg,Flags,HPos,VPos, HBody,VBody) (A0,A1,A2,D0,D1,D2,D3,D4)
-\$00A2 -162	MoveScreen (Screen,dx,dy) (A0,D0,D1)
-\$00A8 -168	MoveWindow (Window,dx,dy) (A0,D0,D1)
-\$00AE -174	OffGadget (Gadget,Ptr,Req) (A0,A1,A2)
-\$00B4 -180	OffMenu (Window,MenuNumber) (A0,D0)
-\$00BA -186	OnGadget (Gadget,Ptr,Req) (A0,A1,A2)
-\$00C0 -192	OnMenu (Window,MenuNumber) (A0,D0)
-\$00C6 -198	OpenScreen (OSArgs) (A0)
-\$00CC -204	OpenWindow (OWArgs) (A0)
-\$00D2 -210	OpenWorkBench ()
-\$00D8 -216	PrintIText (rp,itext,left,top) (A0,A1,D0,D1)
-\$00DE -222	RefreshGadgets (Gadgets,Ptr,Req) (A0,A1,A2)
-\$00E4 -228	RemoveGadgets (RemPtr,Gadget) (A0,A1)
-\$00EA -234	ReportMouse (Window,Boolean) (A0,D0)
-\$00F0 -240	Request (Requester,Window) (A0,A1)
-\$00F6 -246	ScreenToBack (Screen) (A0)
-\$00FC -252	SCreenToFront (Screen) (A0)
-\$0102 -258	SetDMRequest (Window,req) (A0,A1)
-\$0108 -264	SetMenuStrip (Window,Menu) (A0,A1)
-\$010E -270	SetPointer (Window,Pointer,Height,Width, XOFFset, YOFFset) (A0,A1,D0,D1,D2,D3)
-\$0114 -276	SetWindowTitles (Window>windowTitle, screenTitle) (A0,A1,A2)
-\$011A -282	ShowTitle (Screen>ShowIt) (A0,D0)
-\$0120 -288	SizeWindow (Windowmdx,dy) (A0,D0,D1)
-\$0126 -294	ViewAddress ()

-\$012C -300	ViewPortAddress (Window) (A0)
-\$0132 -306	WindowToBack (Window) (A0)
-\$0138 -312	WindowToFront (Window) (A0)
-\$013E -318	WindowLimits (Window,minwidth,minheight, maxwidth, maxheight) (A0,D0,D1,D2,D3)
-\$0144 -324	SetPrefs (preferences,size,flag) (A0,D0,D1)
-\$014A -330	IntuiTextLength (itext) (A0)
-\$0150 -336	WBenchToBack ()
-\$0156 -342	WBenchToFront ()
-\$015C -348	AutoRequest (Window,Body,PText,NText,PFlag, NFlag,W,H) (A0,A1,A2,A3,D0,D1,D2,D3)
-\$0162 -354	BeginRefresh (Window) (A0)
-\$0168 -360	BuildSysRequest (Window,Body,PosText,NegText, Flags,W,H) (A0,A1,A2,A3,D0,D1,D2)
-\$016E -366	EndRefresh (Window,Complete) (A0,D0)
-\$0174 -372	FreeSysRequest (Window) (A0)
-\$017A -378	MakeScreen (Screen) (A0)
-\$0180 -384	RemakeDisplay ()
-\$0186 -390	RethinkDisplay ()
-\$018C -396	AllocRemember (RememberKey,Size,Flags) (A0,D0,D1)
-\$0192 -402	AlohaWorkBench (wbport) (A0)
-\$0198 -408	FreeRemember (RememberKey,ReallyForgot) (A0,D0)
-\$019E -414	LockIBase (dontknow) (D0)
-\$01A4 -420	UnlockIBase (IBlock) (A0)

LAYERS.LIBRARY

-\$001E -30	InitLayers (li) (A0)
-\$0024 -36	CreateUpfrontLayer (li,bm,x0,y0,x1,y1,flags, bm2) A0,A1,D0,D1,D2,D3,D4,A2)
-\$002A -42	CreateBehindLayer (li,bm,x0,y0,x1,y1,flags, bm2) (A0,A1,D0,D1,D2,D3,D3,A2)
-\$0030 -48	UpfrontLayer (li,layer) (A0,A1)
-\$0036 -54	BehindLayer (li,layer) (A0,A1)
-\$003C -60	MoveLayer (li,layer,dx,dy) (A0,A1,D0,D1)
-\$0042 -66	SizeLayer (li,layer,dx,dy) (A0,A1,D0,D1)
-\$0048 -72	ScrollLayer (li,layer,dx,dy) (A0,A1,D0,D1)
-\$004E -78	BeginUpdate (layer) (A0)
-\$0054 -84	EndUpdate (layer) (A0)
-\$005A -90	DeleteLayer (li,layer) (A0,A1)
-\$0060 -96	LockLayer (li,layer) (A0,A1)
-\$0066 -102	UnlockLayer (li,layer) (A0,A1)
-\$006C -108	LockLayers (li) (A0)
-\$0072 -114	UnlockLayers (li) (A0)
-\$0078 -120	LockLayerInfo (li) (A0)
-\$007E -126	SwapBitRastPortClipRect (rp,cr) (A0,A1)
-\$0084 -132	WhichLayer (li,x,y) (A0,D0,D1)
-\$008A -138	UnlockLayerInfo (li) (A0)
-\$0090 -144	NewLayerInfo ()
-\$0096 -150	DisposeLayerInfo (li) (A0)
-\$009C -156	FattenLayerInfo (li) (A0)
-\$00A2 -162	ThinLayerInfo (li) (A0)

-\$00A8 -168

MoveLayerInfrontOf (layer_to_move,
layer_to_be_in_front_of) (A0,A1)

MATHFFP.LIBRARY

-\$001E -30	SPFix (float) (D0)
-\$0024 -36	SPFit (integer) (D0)
-\$002A -42	SPCmp (leftFloat,right,Float) (D1,D0)
-\$0030 -48	SPTst (float) (D1)
-\$0036 -54	SPAbs (float) (D0)
-\$003C -60	SPNeg (float) (D0)
-\$0042 -66	SPAdd (leftFloat,rightFloat) (D1,D0)
-\$0048 -72	SPSub (leftFloat,rightFloat) (D1,D0)
-\$004E -78	SPMul (leftFloat,rightFloat) (D1,D0)
-\$0054 -84	SPDiv (leftFloat,rightFloat) (D1,D0)

MATHIEEEDOUBBAS.LIBRARY

-\$001E -30	IEEEEDPFix (integer,integer) (D0,D1)
-\$0024 -36	IEEEEDPFit (integer) (D0)
-\$002A -42	IEEEEDPCamp (integer,integer,integer,integer) (D0,D1,D2,D3)
-\$0030 -48	IEEEEDPTst (integer,integer) (D0,D1)
-\$0036 -54	IEEEEDPAbs (integer,integer) (D0,D1)
-\$003C -60	IEEEEDPNeg (integer,integer) (D0,D1)
-\$0042 -66	IEEEEDPAdd (integer,integer,integer,integer) (D0,D1,D2,D3)
-\$0048 -72	IEEEEDPSub (integer,integer,integer,integer) (D0,D1,D2,D3)
-\$004E -78	IEEEEDPMul (integer,integer,integer,integer) (D0,D1,D2,D3)
-\$0054 -84	IEEEEDPDiv (integer,integer,integer,integer) (D0,D1,D2,D3)

MATHTRANS.LIBRARY

-\$001E -30	SPAtan (float) (D0)
-\$0024 -36	SPSin (float) (D0)
-\$002A -42	SPCos (float) (D0)
-\$0030 -48	SPTan (float) (D0)
-\$0036 -54	SPSincos (leftFloat,rightFloat) (D1,D0)
-\$003C -60	SPSinh (float) (D0)
-\$0042 -66	SPCosh (float) (D0)
-\$0048 -72	SPTanh (float) (D0)
-\$004E -78	SPExp (float) (D0)
-\$0054 -84	SPLog (float) (D0)
-\$005A -90	SPPow (leftFloat,rightFloat) (D1,D0)
-\$0060 -96	SPSqrt (float) (D0)
-\$0066 -102	SPTieee (float) (D0)
-\$006C -108	SPFieee (float) (D0)
-\$0072 -114	SPAsin (float) (D0)
-\$0078 -120	SPAcos (float) (D0)
-\$007E -126	SPLog10 (float) (D0)

POTGO.LIBRARY

-\$0006 -6	AllocPotBits (bits) (D0)
-\$000C -12	FreePotBits (bits) (D0)
-\$0012 -18	WritePotgo (word,mask) (D0,D1)

TIMER.LIBRARY

-\$002A -42	AddTime (dest,src) (A0,A1)
-\$0030 -48	SubTime (dest,src) (A0,A1)
-\$0036 -54	CmpTime (dest,src) (A0,A1)

TRANSLATOR.LIBRARY

-\$001E -30 Translate (inputString,inputLength,

APPENDIX B

Registers list

Rejestr	Nazwa
\$dff000	BLTDDAT
\$dff002	DMACONR
\$dff004	VPOSR
\$dff006	VHPOSR
\$dff008	DSKDATR
\$dff00a	JOY0DAT
\$dff00c	JOT1DAT
\$dff00e	CLXDAT
\$dff010	ADKCONR
\$dff012	POT0DAT
\$dff014	POT1DAT
\$dff016	POTGOR
\$dff018	SERDATR
\$dff01a	DSKBYTR
\$dff01c	INTENAR
\$dff01e	INTREQR
\$dff020	DSKPTH
\$dff022	DSKPTL
\$dff024	DSKLEN
\$dff026	DSKDAT
\$dff028	REFPTR
\$dff02a	VPOSW
\$dff02c	VHPOSW
\$dff02e	COPCON
\$dff030	SERDAT
\$dff032	SERPER
\$dff034	POTGO
\$dff036	JOYTEST

\$dff038	STREQU
\$dff03a	STRVBL
\$dff03c	STRHOR
\$dff03e	STRLONG
\$dff040	BLTCON0
\$dff042	BLTCON1
\$dff044	BLTAFWM
\$dff046	BLTALWM
\$dff048	BLTCPTH
\$dff04a	BLTCPTL
\$dff04c	BLTBPTH
\$dff04e	BLTBPTL
\$dff050	BLTAPTH
\$dff052	BLTAPTL
\$dff054	BLTDPTH
\$dff056	BLTDPTL
\$dff058	BLTSIZE
\$dff05a	BLTCON0L
\$dff05c	BLTSIZV
\$dff05e	BLTSIZH
\$dff060	BLTCMOD
\$dff062	BLTBMOD
\$dff064	BLTAMOD
\$dff066	BLTDMOD
\$dff068	RESERVED
\$dff06a	RESERVED
\$dff06c	RESERVED
\$dff06e	RESERVED
\$dff070	BLTCDAT
\$dff072	BLTBDAT

\$dff074	BLTADAT
\$dff076	RESERVED
\$dff078	SPRH DAT
\$dff07a	BPLH DAT
\$dff07c	LISAID
\$dff07e	DSKSYNC
\$dff080	COP1LCH
\$dff082	COP1LCL
\$dff084	COP2LCH
\$dff086	COP2LCL
\$dff088	COPJMP1
\$dff08a	COPJMP2
\$dff08c	COPINS
\$dff08e	DIWSTRT
\$dff090	DIWSTOP
\$dff092	DDFSTRT
\$dff094	DDFSTOP
\$dff096	DMA CON
\$dff098	CLX CON
\$dff09a	INTENA
\$dff09c	INTREQ
\$dff09e	ADK CON
\$dff0a0	AUD0LCH
\$dff0a2	AUD0LCL
\$dff0a4	AUD0LEN
\$dff0a6	AUD0PER
\$dff0a8	AUD0VOL
\$dff0aa	AUD0DAT
\$dff0ac	RESERVED
\$dff0ae	RESERVED

\$dff0b0	AUD1LCH
\$dff0b2	AUD1LCL
\$dff0b4	AUD1LEN
\$dff0b6	AUD1PER
\$dff0b8	AUD1VOL
\$dff0ba	AUD1DAT
\$dff0bc	RESERVED
\$dff0be	RESERVED
\$dff0c0	AUD2LCH
\$dff0c2	AUD2LCL
\$dff0c4	AUD2LEN
\$dff0c6	AUD2PER
\$dff0c8	AUD2VOL
\$dff0ca	AUD2DAT
\$dff0cc	RESERVED
\$dff0ce	RESERVED
\$dff0d0	AUD3LCH
\$dff0d2	AUD3LCL
\$dff0d4	AUD3LEN
\$dff0d6	AUD3PER
\$dff0d8	AUD3VOL
\$dff0da	AUD3DAT
\$dff0dc	RESERVED
\$dff0de	RESERVED
\$dff0e0	BPL1PTH
\$dff0e2	BPL1PTL
\$dff0e4	BPL2PTH
\$dff0e6	BPL2PTL
\$dff0e8	BPL3PTH
\$dff0ea	BPL3PTL

\$dff0ec	BPL4PTH
\$dff0ee	BPL4PTL
\$dff0f0	BPL5PTH
\$dff0f2	BPL5PTL
\$dff0f4	BPL6PTH
\$dff0f6	BPL6PTL
\$dff0f8	BPL7PTH
\$dff0fa	BPL7PTL
\$dff0fc	BPL8PTH
\$dff0fe	BPL8PTL
\$dff100	BPLCON0
\$dff102	BPLCON1
\$dff104	BPLCON2
\$dff106	BPLCON3
\$dff108	BPL1MOD
\$dff10a	BPL2MOD
\$dff10c	BPLCON4
\$dff10e	CLXCON2
\$dff110	BPL1DAT
\$dff112	BPL2DAT
\$dff114	BPL3DAT
\$dff116	BPL4DAT
\$dff118	BPL5DAT
\$dff11a	BPL6DAT
\$dff11c	BPL7DAT
\$dff11e	BPL8DAT
\$dff120	SPR0PTH
\$dff122	SPR0PTL
\$dff124	SPR1PTH
\$dff126	SPR1PTL

\$dff128	SPR2PTH
\$dff12a	SPR2PTL
\$dff12c	SPR3PTH
\$dff12e	SPR3PTL
\$dff130	SPR4PTH
\$dff132	SPR4PTL
\$dff134	SPR5PTH
\$dff136	SPR5PTL
\$dff138	SPR6PTH
\$dff13a	SPR6PTL
\$dff13c	SPR7PTH
\$dff13e	SPR7PTL
\$dff140	SPR0POS
\$dff142	SPR0CTL
\$dff144	SPR0DATA
\$dff146	SPR0DATB
\$dff148	SPR1POS
\$dff14a	SPR1CTL
\$dff14c	SPR1DATA
\$dff14e	SPR1DATB
\$dff150	SPR2POS
\$dff152	SPR2CTL
\$dff154	SPR2DATA
\$dff156	SPR2DATB
\$dff158	SPR3POS
\$dff15a	SPR3CTL
\$dff15c	SPR3DATA
\$dff15e	SPR3DATB
\$dff160	SPR4POS
\$dff162	SPR4CTL

\$dff164	SPR4DATA
\$dff166	SPR4DATB
\$dff168	SPR5POS
\$dff16a	SPR5CTL
\$dff16c	SPR5DATA
\$dff16e	SPR5DATB
\$dff170	SPR6POS
\$dff172	SPR6CTL
\$dff174	SPR6DATA
\$dff176	SPR6DATB
\$dff178	SPR7POS
\$dff17a	SPR7CTL
\$dff17c	SPR7DATA
\$dff17e	SPR7DATB
\$dff180	COLOR00
\$dff182	COLOR01
\$dff184	COLOR02
\$dff186	COLOR03
\$dff188	COLOR04
\$dff18a	COLOR05
\$dff18c	COLOR06
\$dff18e	COLOR07
\$dff190	COLOR08
\$dff192	COLOR09
\$dff194	COLOR10
\$dff196	COLOR11
\$dff198	COLOR12
\$dff19a	COLOR13
\$dff19c	COLOR14
\$dff19e	COLOR15

\$dff1a0	COLOR16
\$dff1a2	COLOR17
\$dff1a4	COLOR18
\$dff1a6	COLOR19
\$dff1a8	COLOR20
\$dff1aa	COLOR21
\$dff1ac	COLOR22
\$dff1ae	COLOR23
\$dff1b0	COLOR24
\$dff1b2	COLOR25
\$dff1b4	COLOR26
\$dff1b6	COLOR27
\$dff1b8	COLOR28
\$dff1ba	COLOR29
\$dff1bc	COLOR30
\$dff1be	COLOR31
\$dff1c0	HTOTAL
\$dff1c2	HSSTOP
\$dff1c4	HBSTRT
\$dff1c6	HBSTOP
\$dff1c8	VTOTAL
\$dff1ca	VSSTOP
\$dff1cc	VBSTRT
\$dff1ce	VBSTOP
\$dff1d0	SPRHSTRT
\$dff1d2	SPRHSTOP
\$dff1d4	BPLHSTRT
\$dff1d6	BPLHSTOP
\$dff1d8	HHPOSW
\$dff1da	HHPOSR

\$dff1dc	BEAMCON0
\$dff1de	HSSTRT
\$dff1e0	VSSTRT
\$dff1e2	HCENTER
\$dff1e4	DIWHIGH
\$dff1e6	BPLHMOD
\$dff1e8	SPRHPTH
\$dff1ea	SPRHPTL
\$dff1ec	BPLHPTH
\$dff1ee	BPLHPTL
\$dff1f0	RESERVED
\$dff1f2	RESERVED
\$dff1f4	RESERVED
\$dff1f6	RESERVED
\$dff1f8	RESERVED
\$dff1fa	RESERVED
\$dff1fc	FMODE
\$dff1fe	NO-OP

APPENDIX C

"Asm-One" editor keyboard shortcuts

OPERATIONS ON BLOCKS

AMIGA or CONTROL + B	Select block
AMIGA or CONTROL + B	Select block
AMIGA or CONTROL + C	Copy block
AMIGA or CONTROL + X	Block cut
AMIGA or CONTROL + I	Block insert
AMIGA or CONTROL + U	Deselect the selection
AMIGA or CONTROL + T	Small letters in the block
AMIGA or CONTROL +]	Capital letters in the block

SEARCHING

AMIGA or CONTROL + S
AMIGA or CONTROL + s

Text string search
Find the next occurrence
of the string

AMIGA or CONTROL + R
AMIGA or CONTROL + r

Replace the string
Replace the next part of the
text

JUMPS

AMIGA or CONTROL + !	Mark item 1
AMIGA or CONTROL + @	Mark item 2
AMIGA or CONTROL + #	Check position 3
AMIGA or CONTROL + 1	Jump to position 1
AMIGA or CONTROL + 2	Jump to position 2
AMIGA or CONTROL + 3	Jump to position 3
AMIGA or CONTROL + J	Jump to the comment
AMIGA or CONTROL + and	Jump to a specific line

NAVIGATION

SHIFT + up arrow

SHIFT + down arrow

SHIFT + left arrow

SHIFT + right arrow

SHIFT + A

SHIFT + Z

Previous page

Next page

Beginning of line

End of line

100 lines up

100 lines down

DELETING

CONTROL + Del

Delete part to the end of the line

CONTROL + Back

Delete part to the beginning of the
line

CONTROL + D

Delete the whole line

OTHER

AMIGA or CONTROL + m.
AMIGA or CONTROL + M

Launch macrodefinitions
Enable or disable macro
creating

SELECTED INSTRUCTIONS IN THE COMMAND LINE

Project:

ZS	Program remove
R	Load the program
In	Save the program
ZF	Delete the file
WP	Save settings
= M	Add memory to the workspace
!	Exit from the editor

Editor:

T [line number]	Go to the specified line
B	Go to the end of the file
L [text]	Search for text
ZL [number of lines]	Delete a certain number of lines, starting from the cursor position
P [number of lines]	Print a specified number of lines, starting from the cursor position

Memory:

M [size] [address]

Edit the memory contents

H [size] [address]

Display the memory contents
in hexadecimal

S [size]

Search

C [size]

Copy

Assemble:

A Assembling the program

Disk:

RS [drive]

Read the sector of device

RT [drive]

Read the device path

WS [drive]

Save the sector

WT [drive]

Save the path

Inne:

E	Load an external file
V [path]	Display the contents of directory
>	Set "output" to specified device
? [expression]	Calculate of the value

APPENDIX D

Commands and functions

index

A

Add	115
Addi	153
Allocate	57
And	107, 110, 129
Andi	112
Asr	144
Assemble	70
Assign	41

B

Bmi	151
Bne	148
Bpl	150
Bset	124
Bsr	145

C

Clr	140
Cmp	146
Cmpi	149
Copy	46, 84
Cut	84

D

Divs	122
Divu	122

E

End	81
Eor	134

F

G

H

I

Incdir	174
Include	174
Insert	84
Installer	44, 45

J

Jmp	142
Jsr	144

K

L

Lea	127
Lha	35, 38, 40, 45, 48
Lowercase	84
Lsl	172
Lsr	172

M

Move	112, 127, 136
Movea	109
Muls	121
Mulu	120

N

Not

132

O

Or

133

P

Paste	84
Protect	39

Q

R

Read	75
Read Source	77
Rename	40
Rol	170
Ror	170
Rts	145

S

Search	86
Sub	110, 118
Subi	119, 156
Subq	119

T

Tst

149

U

Unmark	84
Uppercase	84

V

W

Write	75
Write Sourde	77

X

Y

Z

 **AMIGA**.net.pl
